2009

Extending Scojo-PECT by migration based on system-level checkpointing

Peiyu Cai
University of Windsor

NOTE TO USERS

This reproduction is the best copy available.

Extending Scojo-PECT by Migration Based on System-level Checkpointing

By

Peiyu Cai

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2009

Your file Votre référence
ISBN: 978-0-494-57631-1
Our file Notre référence
ISBN: 978-0-494-57631-1

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

In recent years, a significant amount of research has been done on job scheduling in high performance computing area. Parallel jobs have different running time and require a different number of processors, thus jobs need to be scheduled and packed to improve system utilization. Scojo-PECT is a job scheduler which provides service guarantees by using coarse-grain time sharing. However, Scojo-PECT does not provide process migration. We extend the Scojo-PECT by migrating parallel jobs based on system-level checkpointing. We investigate different cases in the Scojo-PECT scheduling algorithm where migration based on system-level checkpointing can be used to improve resource utilization and reduce job response time. Our experimental results show reduction of relative response times on medium jobs over the results of the original Scojo-PECT scheduler and the long jobs do not suffer any disadvantage.

Dedication

To my parents, my aunts

and

Miss. Huajing Yao

Acknowledgements

I would like to take this opportunity to thank my thesis supervisor, Dr. Angela Sodan for her caring, guidance and cooperation throughout my graduate studies. This work presented here would not be impossible without her help and support.

Secondly, I would like to thank Dr. Xiaobu Yuan and Dr. Muscedere for being the committee and spending their valuable time and to Dr. Dan Wu for serving as the chair of the defenses.

Last but not least, I would like to thank all of my friends for all the help and support during the completion of this thesis.

Table of Contents

List of Tables

List of Figures

1. Introduction

In recent years, much research has been done in the High Performance Computing field. A computer cluster system is a group of individual computer nodes usually linked by networks and work as a super computer. Each node in the cluster has one or multiple processors. Nodes also have local memories, local disks, and network devices to communicate and collaborate with other nodes in the cluster system.

In cluster systems, a job can run parallel using multiple processors for a certain period of time, so this makes it important to schedule the jobs of "when" and "where" to run in order to obtain better utilization of the system and better response time for the jobs. The scheduler for the cluster should be designed to obtain high system utilization and reduce the response time of the jobs.

One scheduling category is space sharing. Space sharing [17] partitions processors into groups and each group of processors is assigned exclusively to a single job. The job will run on the assigned processors until finished. This approach is easy to implement but creates fragmentations and the performance is not promising. Another scheduling category is time sharing. The basic idea of time sharing is that multiple jobs share the same resources. Gang scheduling [17] is one of the time sharing approaches which schedules all threads/processes at the same time performs better, but it also suffers from expensive context switches. Jobs are swap/paged out and in on a local machine. So jobs need to restart on the same nodes.

Another possibility to switch running job out to accomplish time sharing is to do checkpointing. In computer systems checkpointing is usually implemented to support rollback-recovery, which aims to recover the system from potential failure. Checkpointing meanwhile allows job schedulers to move jobs or processes, so that the job or the processes of jobs can be re-located and restart at another node. System-level checkpointing requires support from the operating system. It writes the entire image of the running job into storage device. Hence the overhead is high. But system-level checkpointing can be done at any moment by the system. Application-level checkpointing needs the programmer to code checkpoints into their program and only checkpoint part of their data, hence the overhead is reduced. However application-level checkpoints can only be initialized by the user.

The Scojo-PECT [7] scheduler is an existing coarse-grain time sharing scheduler framework, each slice running one type of job and allowing backfilling. At the end of slice, all jobs are preempted and swapped out, then let the next slice type of jobs running.

Scojo-PECT, however, does not support checkpointing and migration. In our thesis we extend Scojo-PECT by adding support of system–level checkpointing and migration, and keep the preemption swaps. This will give our scheduler the flexibility to migrate jobs to different nodes other than those where they were started, hence they can be better packed. We explored the possible cases when and where checkpoints can be done in each coarse-grain time slice. By extending these cases, we expect improvements for the average response time of the Medium type and Long type jobs.

The thesis is organized in the following order: Chapter 2 introduced some important backgrounds; Chapter 3 briefly introduced the Scojo-PECT scheduler; Chapter 4 introduced a review of Checkpoint and Migration; Chapter5 we presented our extension algorithms on the Scojo-PECT scheduler; Chapter 6 presented our experiments and results and Chapter 7 gives a conclusion our thesis.

2. Background Issues

In space sharing, introduced in the previous chapter, the simplest strategy is to schedule jobs in a First Come First Serve order. That means jobs are scheduled to run on the corresponding resources in their submission order. However, the problem for this approach is that jobs are scheduled on the processor dimension only. This will potentially cause fragmentation in the system, which means some processor groups are left idle in a period of time. Because jobs are scheduled in First Come First Serve order, even if the system has enough resources to let a later job run, this job will have to wait for the current running job to complete. As a consequence, average job waiting time and response time are increased. This will hurt the performance.

To address the fragmentation problem, backfilling is studied to fill the system fragmentation and improve system utilization [8][14]. In parallel computing, backfilling means jobs are scheduled to run ahead of their original FCFS order to fill the free resources (fragmentation). There are mainly two types of Backfilling strategies conservative backfilling and EASY backfilling. [34]

If the backfilled job does not delay any of the successive jobs, the approach is called conservative backfilling. Some other aggressive backfilling approaches abandon the no-delay rule. For example: 'EASY' backfilling. This approach relaxes constrains of conservative backfilling by keeping only the first job in the queue not delayed to prevent starvation.

Accordingly, later on jobs might be delayed and hence the response time is affected. As a consequence, this approach could not guarantee good response time. The result varies on different workloads.

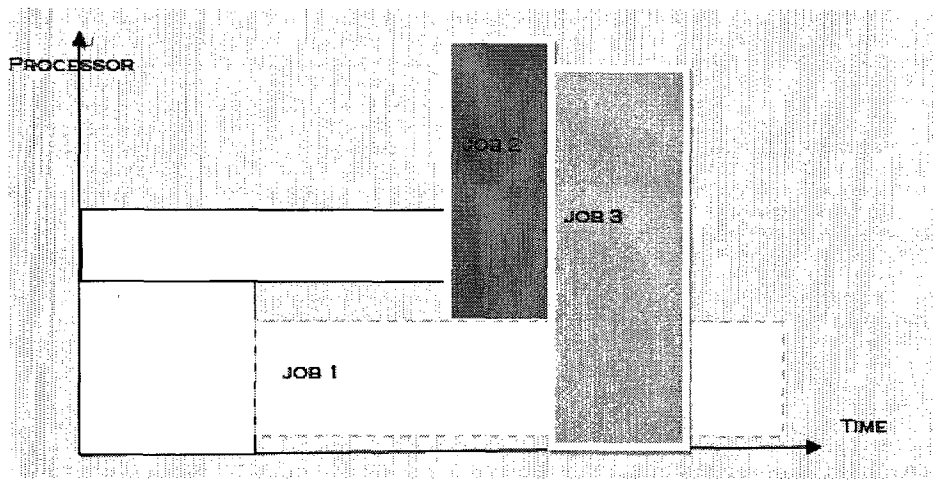


Figure 2.1: Conservative backfilling and EASY backfilling

This figure shows an example of EASY backfilling, Job 1 is an EASY backfilled job, it does not delay Job2 to run, but it conflicts with Job 3, which means Job3 will have to wait until Job 1 is finished. Accordingly, in conservative backfilling, Job 1 will not be backfilled into the space.

Both conservative backfilling and EASY backfilling can improve system utilization significantly. Conservative backfilling improves about 70% system utilization, and still leaves possibility to improve [49]. More over, the average response time is greatly reduced, comparing to the basic First Come First Serve strategy. However, the pitfall of these two approaches are, users have to estimate the jobs' running time accurately, otherwise, the system can not guarantee fairness and correctness of the backfilling. This is also a hotly

studied topic of modern high performance computing research area.

Allowing preemption of jobs is another solution to improve space sharing approaches. Preemption can be used to support priorities, if a job with higher priority comes, low priority jobs may preempt. Preemption can also be used to support long term time sharing. For example, preempt long jobs and run them at night or other non-busy time. The pitfall is that the cost is high and almost all preemptions need support from the operating system.

Preemption can be implemented by suspend/resume in memory or page/swap on disk. However, suspension and resumption in memory may cause memory contention because it requires sufficient system memory to store the currently running job and the preempted jobs as well. Paging and swapping also needs support from the system, which is not always supported. Moreover, preemption requires the preempted job to restart at the same resources. This makes it impossible to re-pack jobs to obtain better response time and utilization.

A more flexible method to switch jobs out is checkpointing. After checkpointing, jobs are able to be migrated; hence it will be possible for jobs to run on different resources. This makes is possible for the scheduler to re-pack jobs better. We will explain checkpointing and migration more in detail.

Checkpointing means to write an image file of the job out including all the information needed to restart the job: all program data, the status of all resources uses such as registers,

opened files and network. Checkpointing is more expensive to do, but it is more flexible than page/swapping.

Once checkpointing is done by the system, migration becomes possible. Migration means to restart checkpointed jobs on different processors. Mere migration keeps the original allocated number of processors. Migration is also implemented by halting the whole job and check pointing. Note that to do migration the job should be in a migration-safe state for example not to be in the middle of updating data or holding a lock. Both checkpointing and migration should not involve system objects, and it is going to be complicated to do on heterogeneous systems. However, static resource allocation courses fragmentation. If the jobs are moldable or malleable, we can apply adaptive resource allocation.

Time sharing is the other general category of scheduling approaches in parallel computing systems. The basic idea of time sharing is to run parallel jobs on time-shared/multi-processing mode.

Combinations of space sharing and time sharing are also introduced such as gang scheduling [21]. Gang scheduling schedules all threads/processes on different processors of a job at the same time. All the processors of the system are time partitioned coordinately, and each time partition can be viewed as a virtual system. Then, all the processes and threads are synchronized into these virtual systems. It also includes more than one job sharing a same time partition. Researches [9] on Gang scheduling show that it significantly reduced average

response time and has better results comparing to Space sharing.

Gang scheduling time slices are set globally and equal. Similar to some other approaches, gang scheduling also suffers from high memory pressure, because it keeps jobs in memory to reduce overhead.

Time sharing without coordination may leads to too many expensive switches or waste of computing resources according to busy waiting. In other words a process of a parallel job may have to wait for other processes of the job. Accordingly, the processor will stay idle for a period of time. Another problem is basic time sharing entails too many expensive system contexts switching. Other strategies have also been investigated such as Scojo-PECT.

The Scojo-PECT [7] scheduler is a coarse-grain time sharing preemptive scheduler framework. It is also globally coordinated like gang scheduling. Scojo-PECT allocates resource shares automatically. More specifically, it explicitly set the resource share distribution per job class dynamically according to the workload, job mix and the administrator's policy. By limiting time slice length into tens of minutes, the preemption/swapping over head is acceptable. Scojo-PECT classifies jobs into job classes according to their running time (currently supports short, medium and long type). Each job type gets an individual slice of time to run. Scojo-PECT preempts jobs at the end of each slice type to free memory space in the system so other jobs can run. We will introduce Scojo-PECT more in detail in next chapter.

The original Scojo-PECT does not support checkpointing; so as a consequence, jobs that were preempted have to restart on the same resources. It is not flexible enough, so we try to extend its flexibility by allowing jobs to migrate based on system-level checkpointing.

Checkpointing technology is a field which has been hotly studied but is still developing. There are no perfect implementations in this area. Our goal is to use checkpointing as a tool to extend the flexibility of space sharing or time sharing.

3. The Scojo-PECT scheduler

In each Scojo-PECT job type slices [7], jobs are scheduled in a first come first serve (FCFS) order. Short jobs and long jobs are found to obtain good response times by employing this approach. Scojo-PECT maintains waiting queues and preemption queues for short, medium and long jobs separately. When a new job comes, the scheduler will classify it by its running time, and then this job will be put in to the waiting queue of its own type, and wait for its time to run. When a time slice runs to its end, all the running jobs will be preempted into their own type preemption queue logically, and the jobs of next time slice will then restart. This approach will reduce memory usage because the jobs can be preempted into disk. Figure 3.1 shows that the time slice intervals changes as time changes.

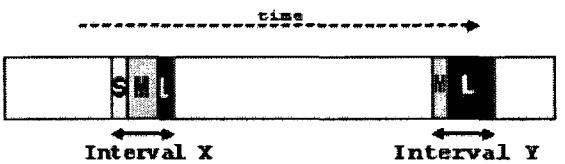


Figure 3.1: Job Type slices and backfilling. [7]

Scojo-PECT reduces fragmentation by applying backfilling. Currently, in our work we are using conservative backfilling policy.

Scojo-PECT additionally supports non-type backfilling to reduce potential fragmentation. Non-type backfilling means that a preempted or waiting job of a different type may get backfilled if it will not delay other jobs. The backfilled jobs will run in the non-type slice until the slice finish, and then this job will be preempted into its own slice type.

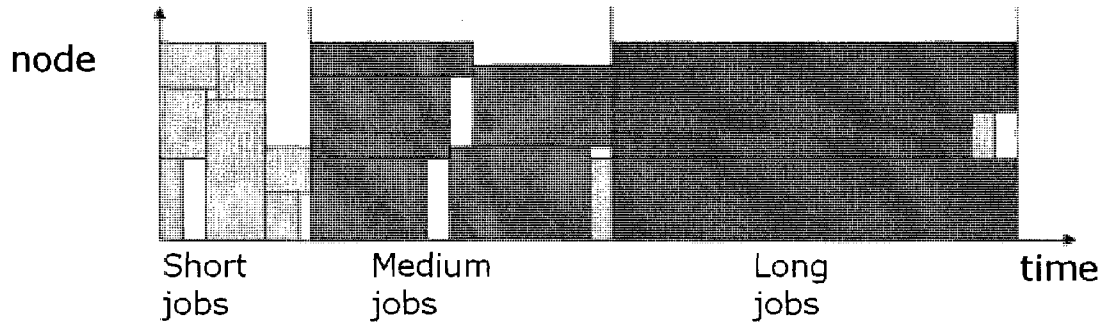


Figure 3.2: Scojo-PECT jobs and backfilled jobs

Scojo-PECT provides dramatically better relative response times than a non-preemptive priority scheduler.

In brief the Scojo-PECT scheduler includes such characters:

1) Classify jobs by job type according to their runtimes (short, medium, long). The three types of jobs are scheduled in separate time slices.

2) Each type of job is scheduled in a FCFS order.

3) Predict job's start time based on conservative backfilling.

4) For each time slice, the PECT scheduler will first resume preempted jobs of the same job type.

5) Then allocate waiting jobs of the same type.

6) Try to backfill (EASY/conservative) after that try non-type backfill.

4. Checkpointing and Migration

Checkpointing is widely used in many kinds of computer systems. In distributed systems checkpointing is normally implemented to support rollback-recovery, which aims to recover the system from potential failure. Checkpointing meanwhile allows job schedulers to move jobs or processes, so that the job or the processes of jobs can be re-located and restart at another node. Hence, by implementing checkpointing, many goals can be accomplished, for example system load balancing, which means when a node of the cluster become overloaded and too "busy", the jobs running on this node can be migrated to other nodes. So checkpointing combined with migration can be used to improve system performance including response time and system utilization.

4.1 Checkpointing mechanism

In this thesis, we use system-level checkpointing and migration as a tool to extend the flexibility of the Scojo-PECT scheduler. System-level checkpointing is a direct and also the most widely implemented checkpointing mechanism is to save the entire state of the job, which is usually called full checkpointing. Because this approach needs support from the computer system, this is also called system-level checkpointing. To implement checkpointing, all the processes must be in a globally consistent state. According to previous research a globally consistent state can be defined as the following: "More precisely, a consistent system state is one in which if a process's state reflects a message receipt, then the state of

the corresponding sender reflects sending that message."[43] Once all the processes of the job are at a consistent state, the processes will then write the state out, on local disk or any kinds of stable storage system e.g. NFS (Network File Systems).

Figure 4.1 shows a typical cluster system with checkpointing supported. A, B and C are nodes of a cluster system, parallel programs run parallel and communicate among the nodes. The application coordinator send signals to the nodes and coordinate the target job that is to be checkpointed to a consistent state, then the checkpoint file will be stored in to the storage system. Accordingly, to restart the job, checkpoint files will be sent to the target nodes and again the coordinator send signals and restart the job at a consistent stage.

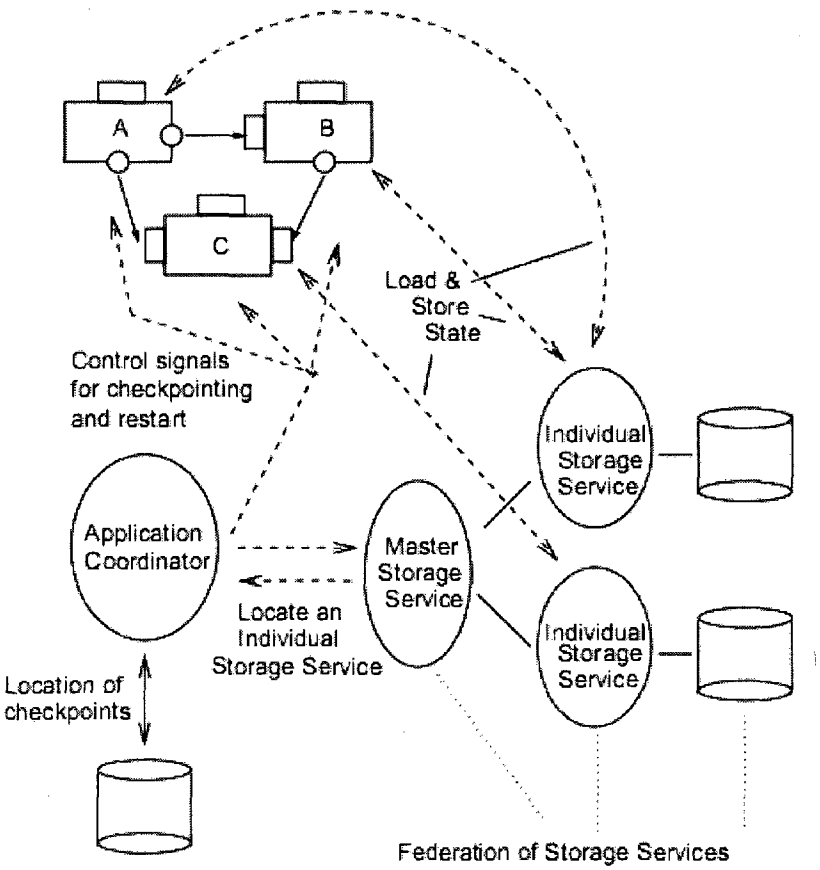


Figure 4.1: A checkpoint-restart system [32]

In some systems that need high reliability, checkpointing has to be done frequently, thus writing the checkpoint file frequently to the storage. This may cost a significant amount of system I/O resource and communication bandwidth. Incremental checkpointing is employed to reduce the checkpoint file size. The system can save only the part that has been modified since the last checkpointing to reduce the file size need to be written out. Then checkpointing can be done frequently and the latency can be hidden. However, incremental checkpointing checkpoint file size is at least as big as the operating system page size. Because of this character, incremental checkpointing can not guarantee that the reduction of the checkpoint files to be significant for every job.

In recent years, computer memory (RAM) price is getting less expensive. This makes it possible to upgrade the memory size of each node of the cluster to a high capacity. As we described above, the major drawback and bottleneck of checkpoint approaches are the system I/O and bandwidth, so saving the checkpoint files in the local memory of each nodes can greatly reduce the system resource cost of checkpointing and restart of jobs.

Another approach is called application-level checkpointing. [44] Application level checkpointing can be supported by external library or supported by compiler. But both need the programmer's effort to initialize checkpointing. The user (programmer) typically needs to define the data set to checkpoint and also define the time interval between checkpoints. In other words, the programmer has to code all these settings into their programs. An

application-level checkpoint file is normally smaller than that of the system-level checkpointing hence the overhead to write the checkpoint file to the storage is lower. However, this approach needs the programmer to know checkpointing structures well and also with this approach it is impossible for the system to initialize a global checkpoint for the entire system.

In our thesis we explored and extended Scojo-PECT scheduler by migration based on system-level checkpointing.

4.2 Checkpointing costs

As we presented above, the cost of checkpointing of a parallel job in a cluster system typically contains two parts:

1) To reach the consistent state, the system needs to coordinate all the processes, this takes a certain period of time. We denote it Coord.

2) Once all the processes of the job are in a consistent state, the state image of each process will be written out on local disk or in the memory or on a NFS server. At system level the image size is equal to the memory footprint size.

We define the image file size that needs to write out as IS and the bandwidth to write out bandwidth. So the formula of the checkpointing cost is:

$$Cp\ cost = Coord + IS/bandwidth$$

The coordinating time of processes is not dependent on the number of nodes and it is not the main latency. By comparing the cost of in-memory and on-disk checkpointing, the cost of in memory checkpointing is under 1 second, this includes coordination of processes and write into memory. Hence Coord is less than 1 second roughly 0.4 sec or even less. [47][50]

The bandwidth to write the process image depends on the implementation. According to [49] the bandwidth to write on the local disk is around 55~98MB /s to local disk in 2005, so it is reasonable to believe that it can commonly be 70MB/s in 2009. If write into the memory it can be as fast as 1G/s. If store to a remote NFS server, the bandwidth will depends on the network connection. A grid computing system in Canada called Sharcnet [*www.sharcnet.org*] has 10Gbps/1Gbps dedicated connection between all clusters within the grid computing system itself. So 70Mb/s is also reasonable for remote NFS server storage.

From the above information, the formula of the checkpointing cost we use in this thesis would be:

$$CP\ Cost = IS/70 + 0.4$$

Accordingly, the in memory checkpointing cost would be

$$CP\ Cost = IS/1024 + 0.4$$

4.3 Migration and restart mechanism and cost

Since checkpointing stores a consistent state of a job, by transporting the state of process and restarting it in a different node, we can achieve process migration.

Migration time is linear with the checkpointing image size. Mainly it costs the system to write the image to the target nodes. And it is independent of the number of the processes in the job [47]. Process restart itself costs a very short time. In some existing unix/linux system, the system use mmap() function in Unix system instead of writing all the data into the memory to restart the process.[48]

The cost to migrate and restart the job can be defined as follow:

$$MR\ cost = restart\ cost + IS/bandwidth$$

As we discussed above, the restart cost is very small [47] it is reasonable to believe it is within 0.4 second. IS is the checkpoint file image size. Bandwidth is the achievable transportation bandwidth to transport the image file to the target nodes. In [51] the bandwidth is 22Mb/s in year 2005, it is reasonable to believe that the bandwidth now can be 30Mb/s. So the formula to do process migration and restart is:

$$MR\ cost = IS/30 + 0.4.$$

Then we can obtain the total Checkpointing and Migration restart cost:

$$CMR\ cost = IS/21 + 0.8$$

Accordingly, in memory checkpoint restart would be:

$$MR\ cost = IS/1024 + 0.4$$

Then the total Checkpointing and Migration restart cost would be:

$$CMR\ cost = IS/512 + 0.8$$

5. Extending Scojo-PECT by migration based on system-level checkpointing

The Scojo-PECT scheduler is a coarse-grain time sharing preemptive. It combines time sharing and space sharing approaches to reduce response time and increase system utilization.

5.1 Our assumptions

Our extension on Scojo-PECT scheduler is based several assumptions as follows.

- Each node contains only one processor, local memory, operating system and sufficient local disk space.
- All jobs are neither moldable nor malleable. That means once submitted the number of processors they need are fixed and can not change. In parallel computing, Moldable job means the number of processors the job needs can be modified before starting to run. Malleable job means the number of processors the job needs can be modified both before starting to run and while it is running.[31]
- Users will have full knowledge of their job and are able to provide the number of processors their jobs need. Also, the users will provide to the system with their estimated running time of their jobs. This assumption used to be hard to accomplish. However, recently, more and more commercial scientific computing softwares are released. More and more researchers chose to use softwares to run tests on clusters instead of programming by their own. These softwares can collect these job characters for the user

automatically or offer such interfaces to the user based on the input of the user's problem e.g. "Gaussian 09", "Amber".

5.2 Selected Metrics

The following are the metrics we employed:

- Average Relative Response time: The time period from the moment the job submitted to the cluster system to the moment that the job completed its running is called response time of the job. Relative Response time is defined as: "response time /pure running time without time slicing". This metric represents more on behalf of the user.
- Another metrics is P75, it denotes the highest bound of 75% jobs' relative response times.
- Similarly P95 denotes the highest bound of 95% jobs' relative response times.

5.3 Extended Cases

Our efforts and work to extend Scojo-PECT by adding migration based on system-level checkpointing can be categorized into the following four basic cases:

5.3.1 Case one: Move job to continue in next non-type slice

In Scojo-PECT scheduler, if a job can not finish at the end of it's own type slice, it will be preempted and store into the preemption queue and wait for its own type of slice again then restart on exactly the same processors it was running. However, if there are enough free

resources in the next slice for this job to run, because this job was preempted, it has to start on the same processors, the free resource will be wasted and stay idle.

As illustrated in Figure 5.1 below, A job of Job Type A can not finish within its own slice type, it need some more time to finish, but at the end of Slice Type A, this job must be preempted and wait for Type B slice before it gets a chance to run again. However, we can discover that in job Type B slice, there are enough free nodes (processors) to run this job, because the original Scojo-PECT does not support checkpointing, these resources will be wasted.

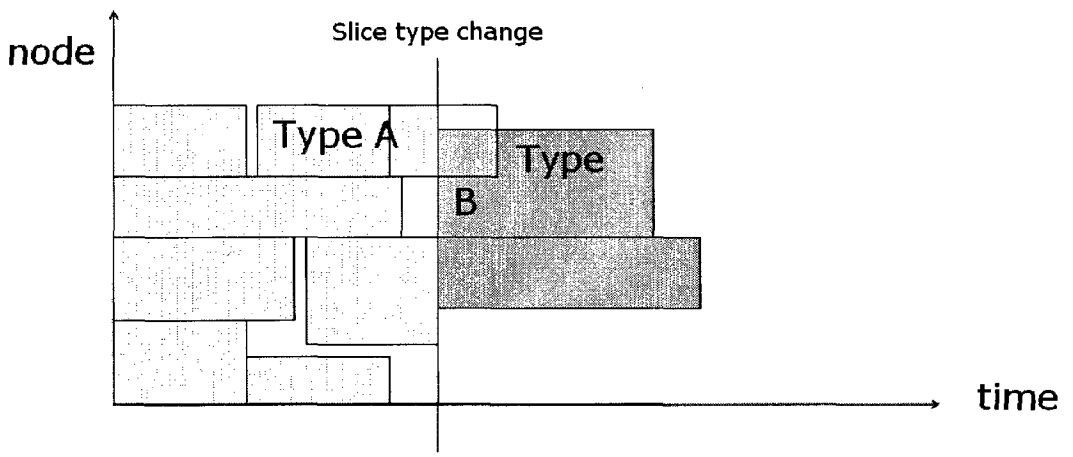


Figure 5.1: Move job to continue in next non-type slice(before)

To make use of the free processors in the next slice, we can apply a checkpoint on the job and then we restart the job to the free processors in the next slice. In Figure 5.2 the Type A job is checkpointed. The deeper color part is the overhead to do the checkpoint. Then the checkpointed job is migrated to Type B slice, the deeper colored part represents the overhead to transfer the checkpoint file and restart the job.

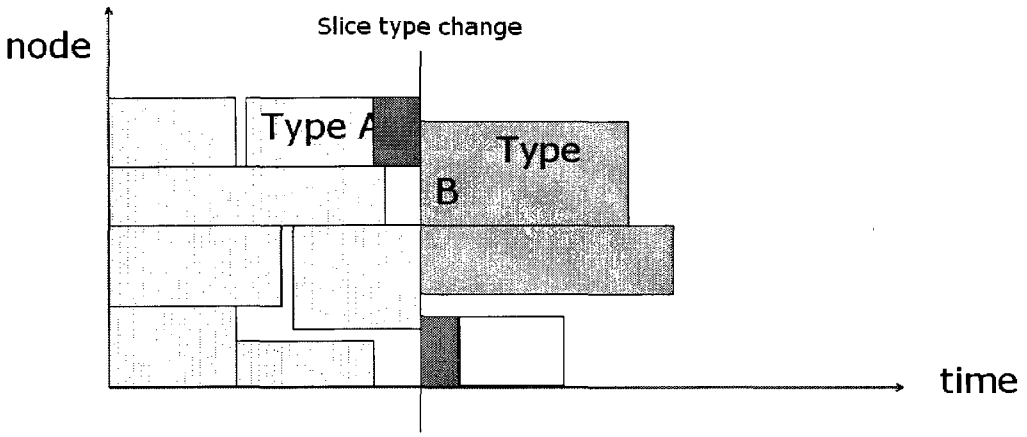


Figure 5.2: Move job to continue in next non-type slice(after)

The scheduling Algorithm in presented as follow:

```
Case (SliceBeginEvent)   { // CP is the abbreviation of checkpoint
     For(jobsInRunningQueue){
          If (jobRemainRunningTime > sliceRemainingTime){
           If(thereIsResourceCollisionInNextSlice){
                if(SliceFinishTime – currentTime > CPoverhead(job) )
                add(job, CandidateList);
           }
     }
For(allJobsInCandidateList){
     makeCombinationsOfCP;
}
     For (allCpCombinations){
          compareCpGain(combination);
          Return(TheCombinationWithHighestGain);
     }

     insertCpEvents(combination);
     }
```

5.3.2 Case two: Move job to make space for non-type backfill job

This case addresses another situation. While a time slice is running, a job is finished, then the processors allocated to it will be set free and ready for other jobs to run. At this point, the Scojo-PECT scheduler will first try to run own type waiting jobs and then try to backfill jobs from other types. However, after all the checking, there is still a situation that the number of free processors is enough for a preempted job of another type, but only because the job is a preempted job it has to restart on the same processors it was running, and some of these processors are occupied by another running job, hence this preempted job can not be backfilled and run in current slice. In Figure 5.3 we can see in job Type A, a job finished in the middle of this current slice, accordingly its processors are set free. A preempted Job 2 of Type B can use these processors but it is blocked by Job 1.

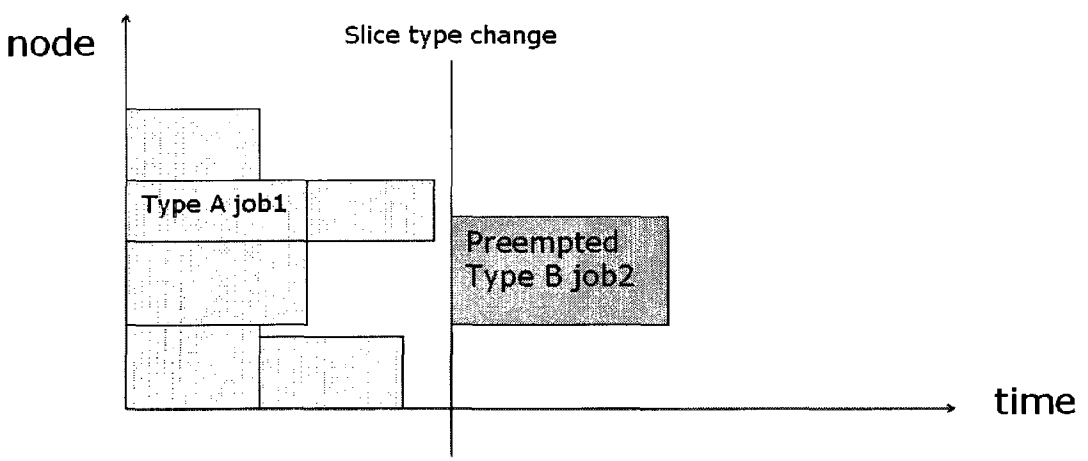


Figure 5.3: Move job to make space for non-type backfill job(before)

To make use of these system resources, we can apply a checkpoint on the job that is blocking the preempted job, and migrate it to some other processors that will not block the preempted job from being backfilled. In Figure 5.4 job one is checkpointed and migrated to some other

processors and the preempted Type B Job 2 is non-backfilled and run in slice Type A. Hence this will improve the response time of Job 2.

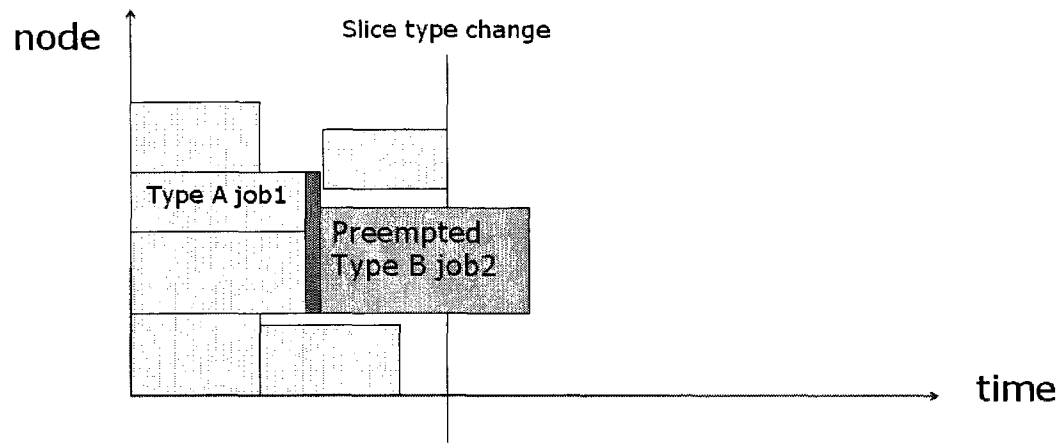


Figure 5.4: Move job to make space for non-type backfill job(after)

The scheduling algorithm is presented as follow:

```
Case(jobFinishEvent){
Schedule() // try schedule jobs using the existing scheduler

for (alljobsInPreemptionQ){
    if (currentFreeNodes>jobNodes ){
    getJobsThatAreBlockingBackfill(); // jobs that blocks the preempted job
    if(backFillGain > CheckpointMigrationCost){
        Migrate(blockingJobs);
        Non-typeBackfill(preemptedJob);
        }
    }
}
```

5.3.3 Case three: New job non-type backfill with migration at the end of slice to avoid conflictions

In the original Scojo-PECT scheduler, when a job finishes, the scheduler will try to non-type backfill jobs from other types. If a non-type job can fit into the free processors in the current slice, but this job can not finish within the slice and needs to run on its own slice type, then, the scheduler will check again if this job will conflict with the preempted jobs in next slice, if it will, the job will normally not be backfilled. In Figure 5.5 in Type A slice, we can see there are enough free processors for a new Type B job 1, however, this job can not finish within slice Type A, and moreover, this job conflicts with a preempted Type B job 2, so hence this Job 1 will not be non-type backfilled in the original Scojo-PECT scheduler without checkpointing and migration.

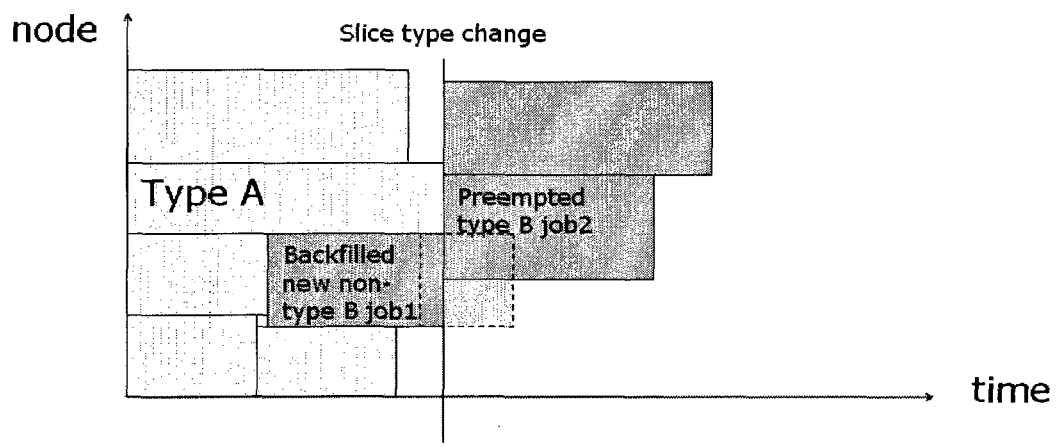


Figure 5.5: New job non-type backfill conflicts (before)

To extend the original scheduler, we allow the non-type waiting job backfill, then at the end of the slice before the slice switch, we checkpoint the non-type backfilled job and migrate the

job to processors that are free in its own slice type and let it continue to run. In Figure 5.6, the Type B job 1 is non-type backfilled and checkpointed at the end of slice Type A then in its own type slice B, it is migrated to the free processors that will not conflict with the preempted job 2.

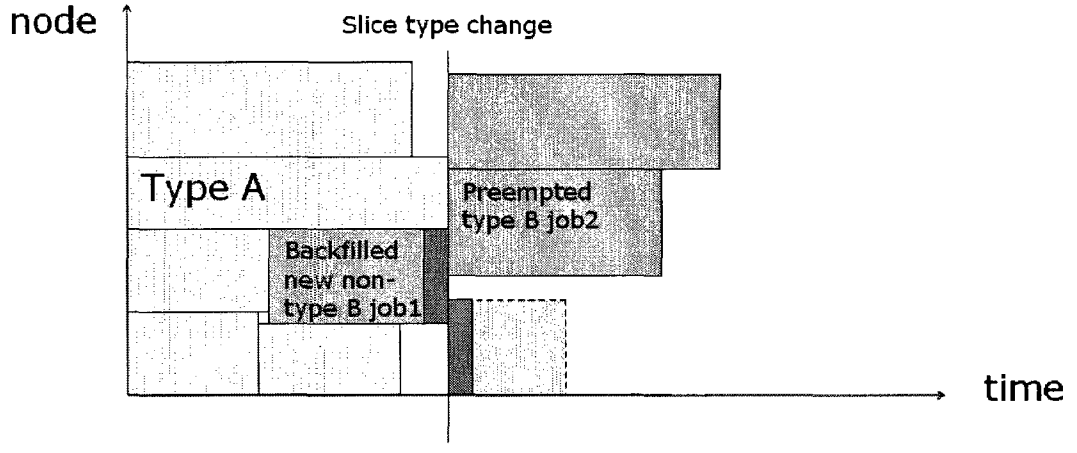


Figure 5.6: New job non-type backfill with migration at the end of slice to avoid conflicts(after)

The algorithm to handle this case is as follows:

```
Case (jobFinishEvent) { // where processors are released.
    schedule();// let the original scheduling algorithm run first
    for (waitingjobsofNextType){
        if(fitInCurrentSlice && cannotFinishWithinCurrentSlice){
            if (conflictsWithJobpreemptionQueue) {
            if(CheckpointMigrationCost < checkpointMigrationGain)
                Non-typeBackfill(job);// let the job run
                addCPevent(job); // let job cp before slice switch
        }
    }
CaseCPevent{ // when it is time to checkpoint
        Checkpoint(job);
        Migrate(job,freeNodes);//migrate the job into free nodes
    }
```

5.3.4 Case four: Remove non-type backfilled job if new job of own type arrives

In the original Scojo-PECT scheduler, if a job is non-type backfilled into another type slice, this job will continue to run till the end of this slice. This makes a problem, if a new job of the current slice type arrives and at this moment there are not enough free processors in the system to run this job, it will be delayed and keep waiting till its own slice come again. This will hurt the response time of this job when improving the response time of the non-type backfilled job. Figure5.7 shows at Type A slice a Type B job was non-type backfilled and running, a new job of type A arrives when the backfilled job hasn't finish yet.

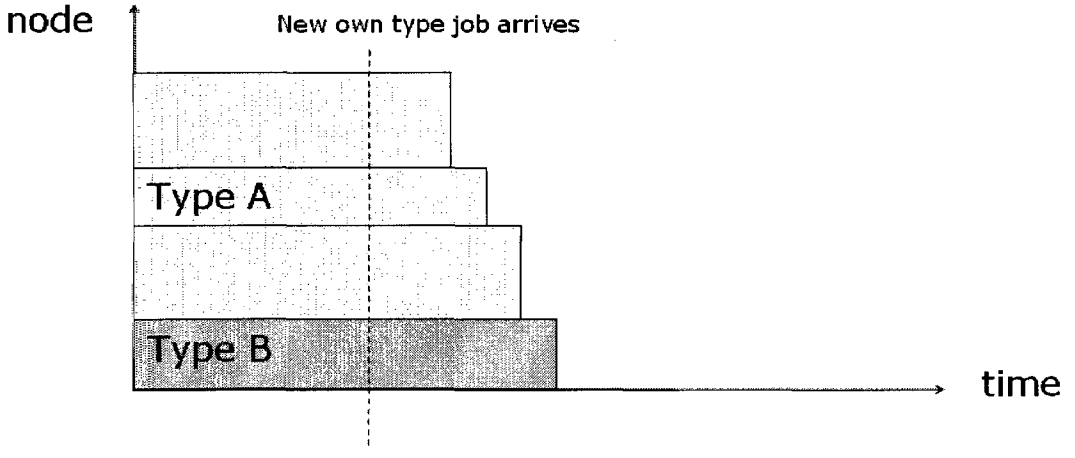


Figure 5.7: Remove non-type backfilled job if new job of own type arrives(before)

We extend this case by giving the non-type backfilled job a checkpoint and put it into the preemption queue of its own type. Then the new arrived job can get a chance to run. In Figure 5.7 we can see that the non-type backfilled Type B job is checkpointed and a newly arrived job of Type A starts to run on the processors yielded by the checkpointed job.

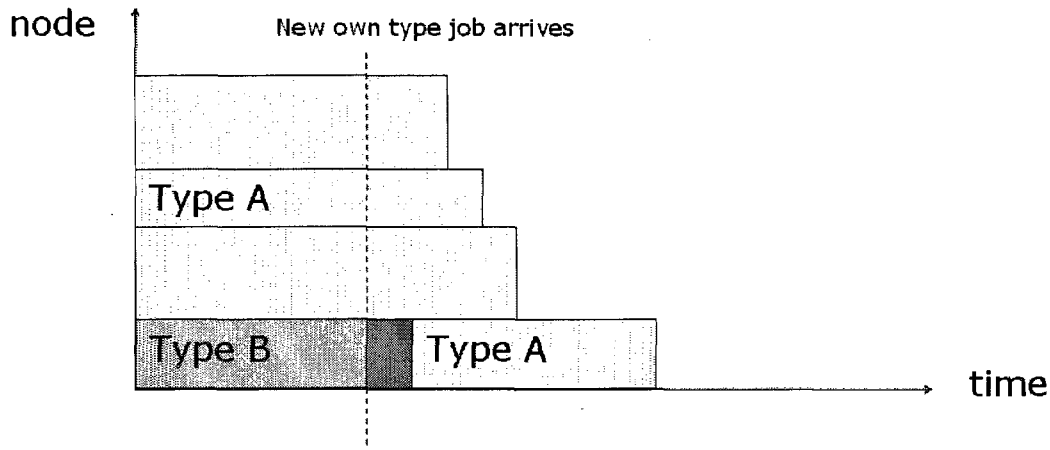


Figure 5.8: Remove non-type backfilled job if new job of own type arrives (after)

The following is the algorithm of this case.

```
Case (jobSubmitEvent){
Schedule(WaitingJobs);//try to schedule with the original scheduler

For (jobsInRunningQueue){
    if (jobType != currentSliceType
        && enoughSpaceInNextSliceFor(job)
        && currentFreeNodes + jobNodes >= waitingJobNodes){
            Checkpoint(job);
            Start(waitingJob);
        }
    }
}
```

5.4 Sub Cases of Extended Cases

Although we extended the original Scojo-PECT scheduler with the four basic cases, as we implement we have to face more sub-cases that may occur. The following are the sub-cases we have to deal with in a real cluster system.

5.4.1 Move one job which can stay on same resources

This is the most straight forward case in our basic case one. Job 1 is checkpointed and migrated to another slice, then run and finish with in the next slice type. In Figure 5.9 job 1 is checkpointed in its own type slice and migrated into the next type slice, then it ends before the next slice is over.

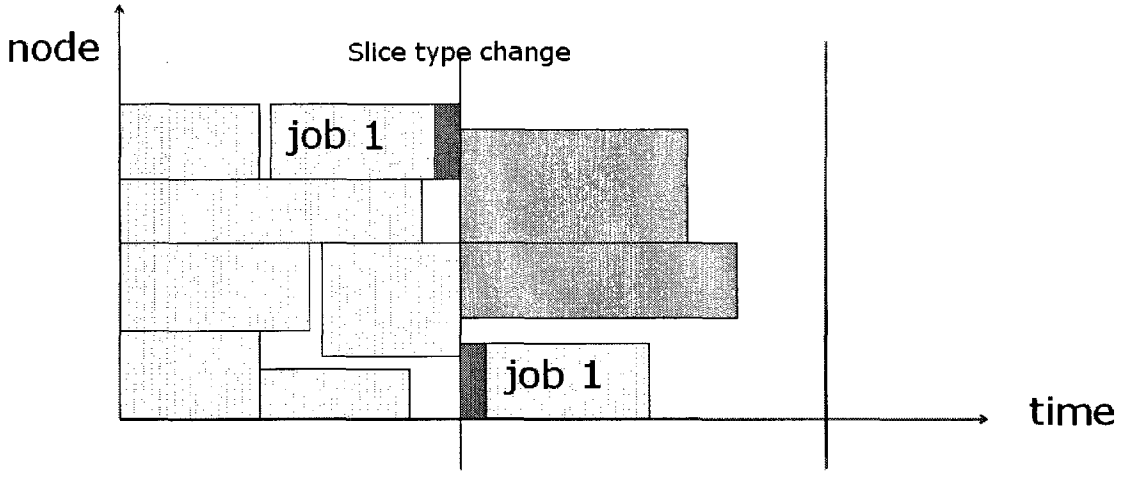


Figure 5.9: Move one job which can stay on same resource

5.4.2 Move one job which can not stay on same resources

In this case a job is checkpointed and migrated to another type slice as the pervious case, but the migrated job is too long to finish within the next type slice. This may cause processor conflicts if there are preempted jobs of its own type occupied the processors. So we have to checkpoint the job again and migrate back to its own position when its own type slice comes. Accordingly the overhead is again increased and have to be recalculated. In Figure 5.10 job 1 of type A is checkpointed and migrated to Type B slice. But it can not finish when Type B

slice ends, so it is checkpointed again and migrated back to its own position after Type C slice when Type A slice comes again.

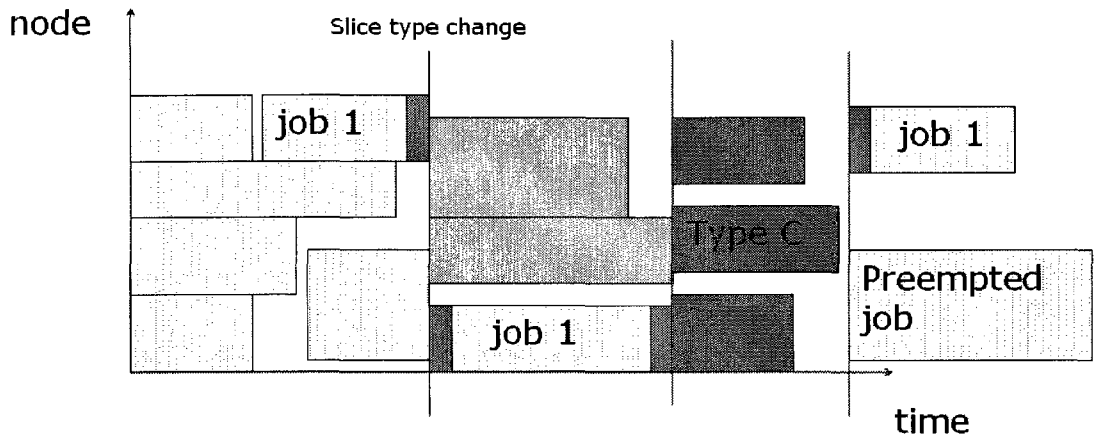


Figure 5.10: Move one job which cannot stay on same resources

5.4.3 Move multiple jobs to use resources in next slice

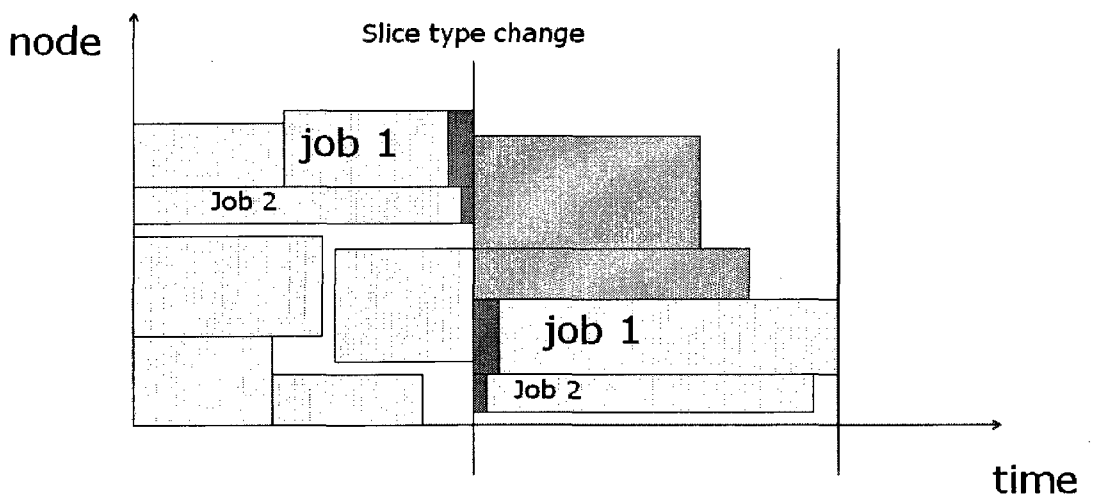


Figure 5.11: Move multiple jobs to use resources in next slice

Another sub-case that may occur is that when we try to move job to continue in next non-type slice, there may be more than one job that can fit into the free processors and

continue to run, we will have to evaluate the gain and cost of all the combinations of the fixable jobs and make the decision which job or combination of the jobs should be checkpointed and migrated. In Figure 5.11 job1 and job 2 can be checkpointed migrate together into next type slice and continue to run.

5.4.4 Move multiple jobs to avoid conflict

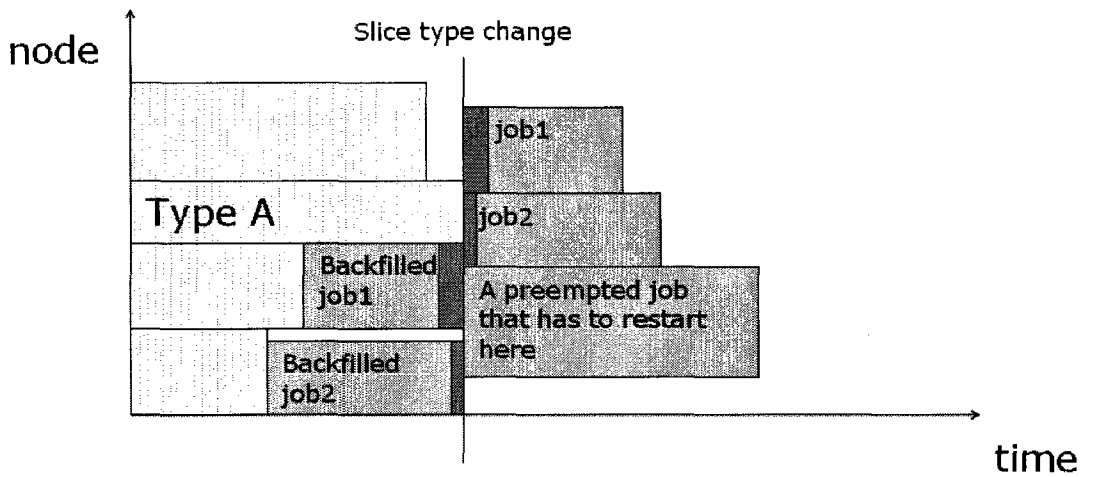


Figure 5.12: Move multiple jobs to avoid conflict

Similarly if there are multiple jobs that can be backfilled into a non-type slice and then can avoid conflict with preempted jobs of its own type, we will make the decision which job or which combination of jobs we should checkpoint and migrate. In Figure 5.12 job 1 and job 2 are both non-type backfilled into type A then at the end of slice Type A they are both checkpointed then migrated in their own slice type to avoid conflicts with an existing preempted job.

5.5 Utilization Gain Calculation

The Utilization Gain Calculation module is an important part of our extension on the Scojo-PECT scheduler. Whenever we want to checkpoint and migrate a job for any of the cases we introduced above, we need to first calculate the cost and the gain of the checkpoint and migration, only if the gain is larger than cost we can continue to checkpoint and migrate jobs.

We select which job(s) based on extra executed time during non-type slice minus overhead for the procedure of checkpointing and migration of the job(s).

If using disk storage systems, the cost of checkpoint and migration can be simulated to be:

Checkpoint cost = Coordinate cost + ImageSize/bandwidth*I*

= 0.4 + (job memory footprint size) / 70 (Mb/s)

Migration cost = restart cost + ImageSize/bandwidth*O*

= 0.4 + (job memory footprint size) / 30 (Mb/s)

If the job(s) can finish within next non-type slice its Gain can be calculated as:

Gain = JobRemainningRuntime – (CheckpointCost + MigrationCost)

If the job(s) cannot finish within next non-type slice, that means this job will be first migrate to next non-type slice and execute, then at the end of the slice, do another checkpointing and

migrated back to its own type slice to avoid potential conflicts .

Hence the gain would be: extra executed time in next non-type slice minus the overhead to do the checkpoint and migration twice including migrating back:

Gain = LengthOfNextNon-typeSlice – 2*(CheckpointCost + MigrationCost)

5.6 Making decisions among checkpointing candidates

Although we calculate the gain and cost before making the decision which job or combination of jobs should be checkpoint and migrated to obtain the maximum gain, this is not enough for a real cluster system. Modern cluster systems contain hundreds even thousands of processors, this allows many jobs to run simultaneously. As a result, there can be a large number of jobs that can be checkpointed and migrate. Then, calculating the gain and cost of all possible job combinations could be very costly and became unaffordable. To solve the problem we apply heuristics to reduce the computation load.

In our extension Case 1, if we have more than one job that can fit into the free processors in the next non-type slice, we first calculate the cost and extra running time of individual jobs, if the extra running time is larger than the cost, this job will be added to a candidate list. Then we check possible job combinations with a maximum of 4 elements of all the candidates in the list. And then we perform checkpointing and migration for the combination of jobs with the highest gain.

In our Case 2 once a job finishes in current slice, if the processors freed can fit more than one preempted job to be non-type backfilled, we first calculate the extra running time of each preempted job and then calculate the cost to migrate corresponding jobs need to be checkpoint and migrated. Then we non-type backfill the job with the highest gain, checkpoint migrate corresponding jobs and then see if the rest of processors can fit the job with the second largest Gain and then repeat until all the candidates are checked or no more free processor left in next non-type slice.

In our Case 3, it is very similar to case one. If the free processors in the current type slice are enough for multiple non-type waiting jobs we select the one with the highest gain and then try to fit more jobs with the same procedure.

In our Case 4, because the newly arrived job of own type has a higher priority, so it is fair that we chose the first newly arrived job, checkpoint the corresponding non-type job(s) and let the new job run. If there are multiple non-type backfilled jobs running, we checkpoint the job(s) using less processors if it does not delay own type jobs.

By merging our extension and heuristics into the original Scojo-PECT scheduling algorithms, our extension does not increase the algorithm complexity of the Scojo-PECT scheduler. In our simulation, the running time of our extension for 10,000 jobs is increased about 15% than that of the original Scojo-PECT simulation.

6. Experiments and Results

6.1 Experimental Set-up

Our experiments and evaluation are based on a discrete event simulator. The simulation input data is generated by an external library in the Lublin-Feitelson [17] model. This model simulates and generates workloads based on tens of thousands real jobs of real cluster traces from the following three systems [17].

- "San-Diego Supercomputer Center Intel Paragon machine. This system has 416 nodes and the log covers all 1995 and 1996."
- "1024-node Connection Machine CM-5 installed at Los-Alamos National Lab. The log is from January through September 1996"
- "100-node IBM SP2 machine at the Swedish Royal Institute of Technology in Stockholm. In the Period of October 1996 to August 1997"

Based on Lublin-Feitelson model, we generated workloads with the following randomization seeds: 7,31,73,13,71. Each set of workload has 10,000 jobs and these jobs are simulated to run on a cluster system which has 128 nodes. Different workload will have different impacts on the result of our approach. The following Table 6.1 describes the most important characteristics of the workloads with different seeds.

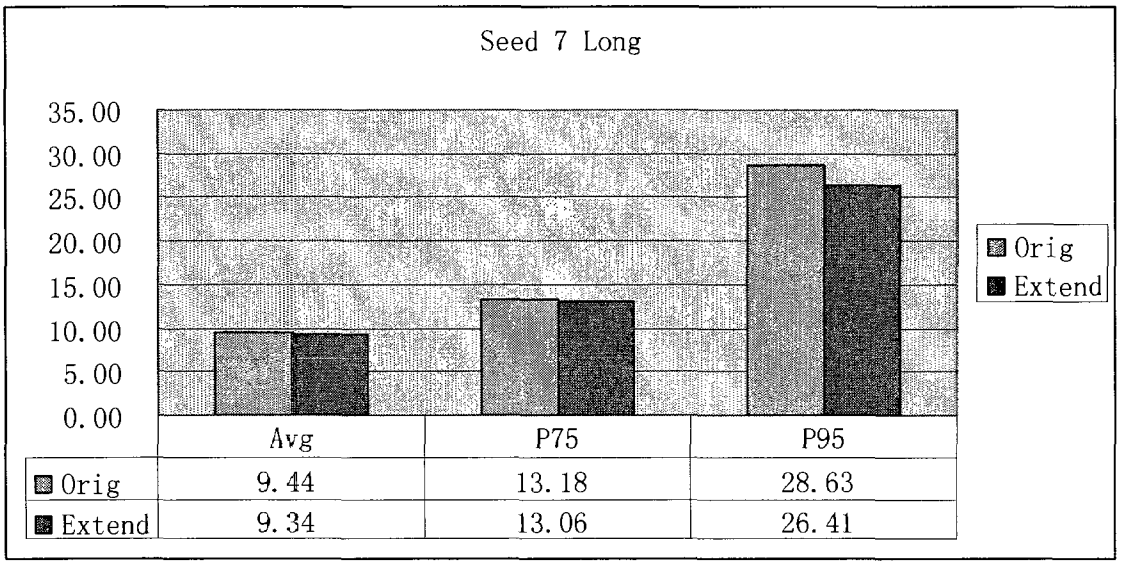
Table 6.1: Characteristics of generated workloads

SEED	Percentage of Job types			Average Job size			Average Inter-Arrival Time(sec)	Overall Utilization
	Short	Med	Long	Short	Med	Long		
7	64.31	19.68	16.01	8.63	17.35	20.87	840.0	0.8037
13	63.57	19.18	17.25	8.53	16.98	19.70	832.0	0.7944
23	64.77	19.17	16.06	8.70	16.60	20.38	1038.0	0.6215
31	63.09	20.18	16.73	8.72	16.67	20.43	860.0	0.7688
71	64.03	19.39	16.58	8.68	16.65	19.02	810.0	0.7643

6.2 Experimental Results

The following figures are the comparison of relative response times between the original Scojo-PECT running workloads with different seeds and the result of extended Scojo-PECT scheduler with our Checkpoint and Migration extension.

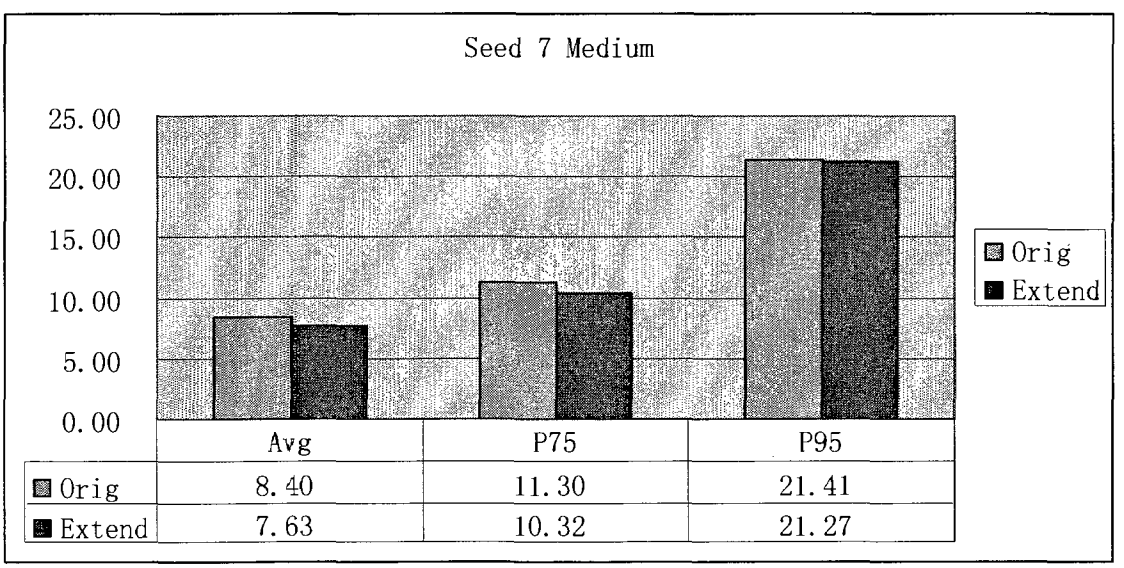
Avg denotes the average relative response time for all the jobs. P75 represents the average relative response time for 75% of all the jobs. Similarly P95 is the average relative response time for 95% of all the jobs.



	Avg	P75	P95
Orig	9.44	13.18	28.63
Extend	9.34	13.06	26.41

Figure 6.1: Seed 7 Long jobs relative response time comparison

Figure 6.1 show the comparison of relative response times for Seed 7 Long jobs between the Original result and our Extension.

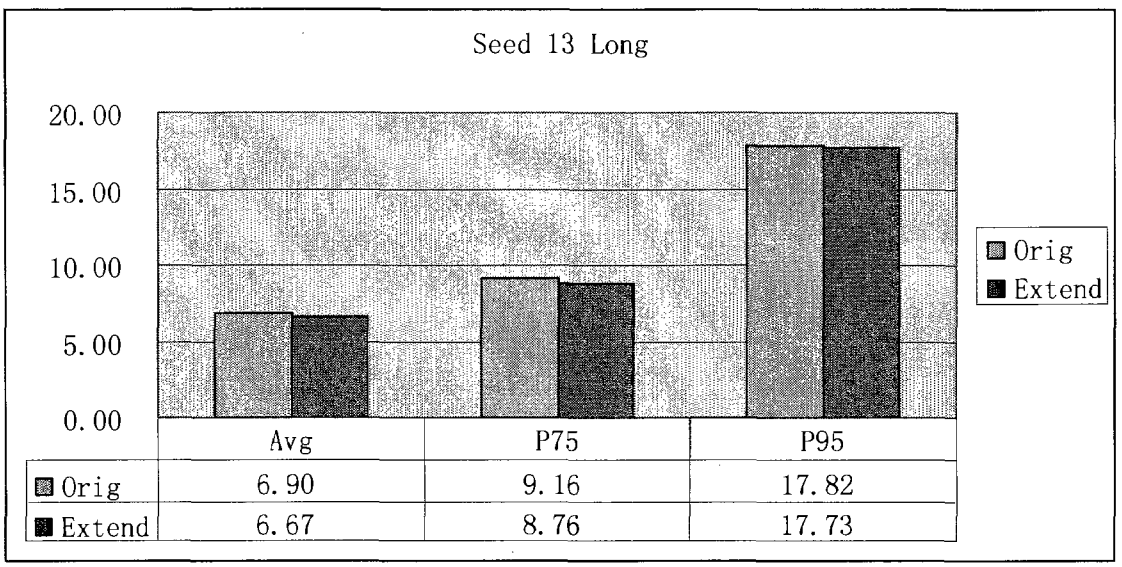


	Avg	P75	P95
Orig	8.40	11.30	21.41
Extend	7.63	10.32	21.27

Figure 6.2: Seed 7 Medium jobs relative response time comparison

Figure 6.2 shows the comparison of relative response times for Seed 7 Medium jobs between

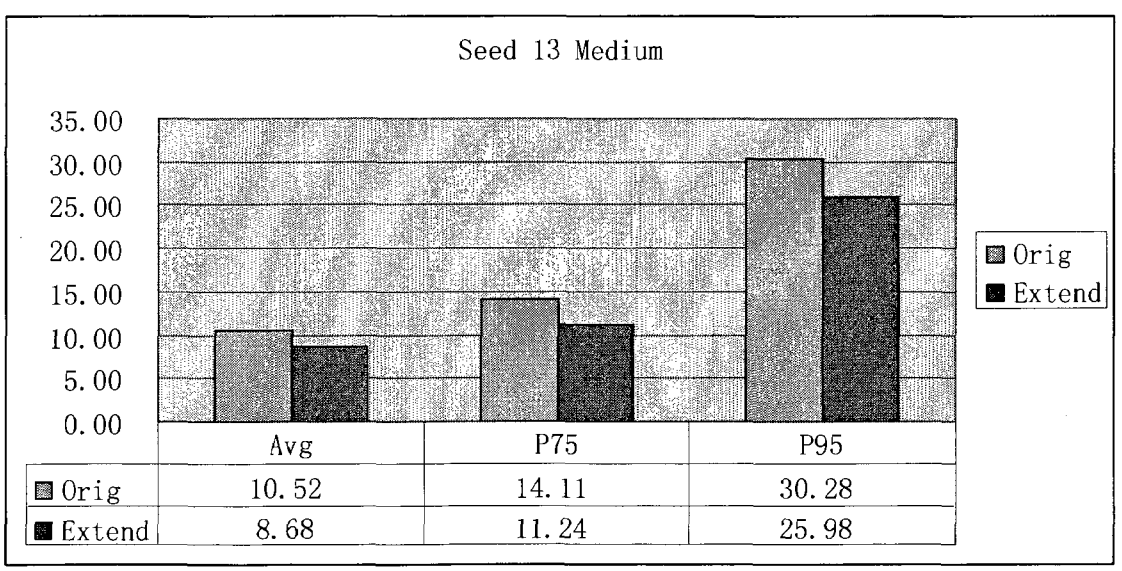
the Original result and our Extension.



	Avg	P75	P95
Orig	6.90	9.16	17.82
Extend	6.67	8.76	17.73

Figure 6.3: Seed 13 Long jobs relative response time comparison

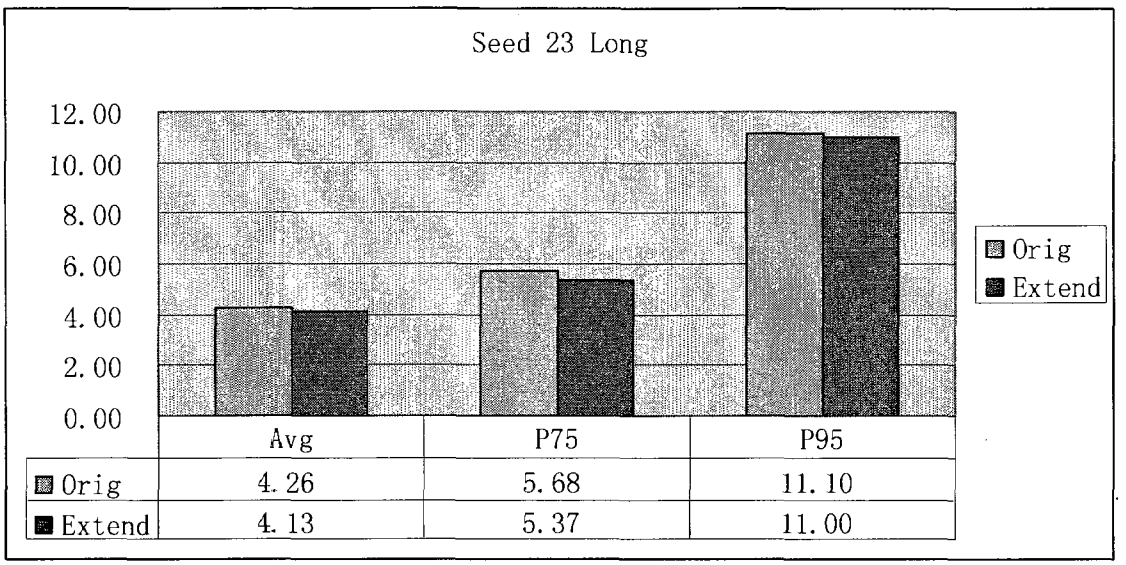
Figure 6.3 shows the comparison of relative response times for Seed 13 Long jobs between the Original result and our Extension.



	Avg	P75	P95
Orig	10.52	14.11	30.28
Extend	8.68	11.24	25.98

Figure 6.4: Seed 13 Medium jobs relative response time comparison

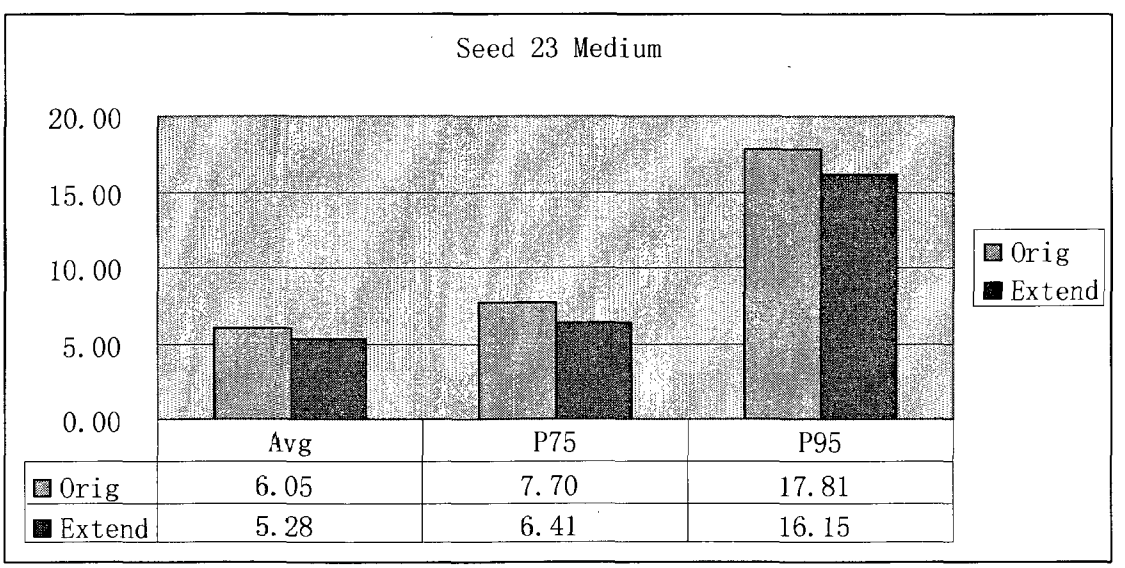
Figure 6.4 shows the comparison of relative response times for Seed 13 Medium jobs between the Original result and our Extension.



	Avg	P75	P95
Orig	4.26	5.68	11.10
Extend	4.13	5.37	11.00

Figure 6.5: Seed 23 Long jobs relative response time comparison

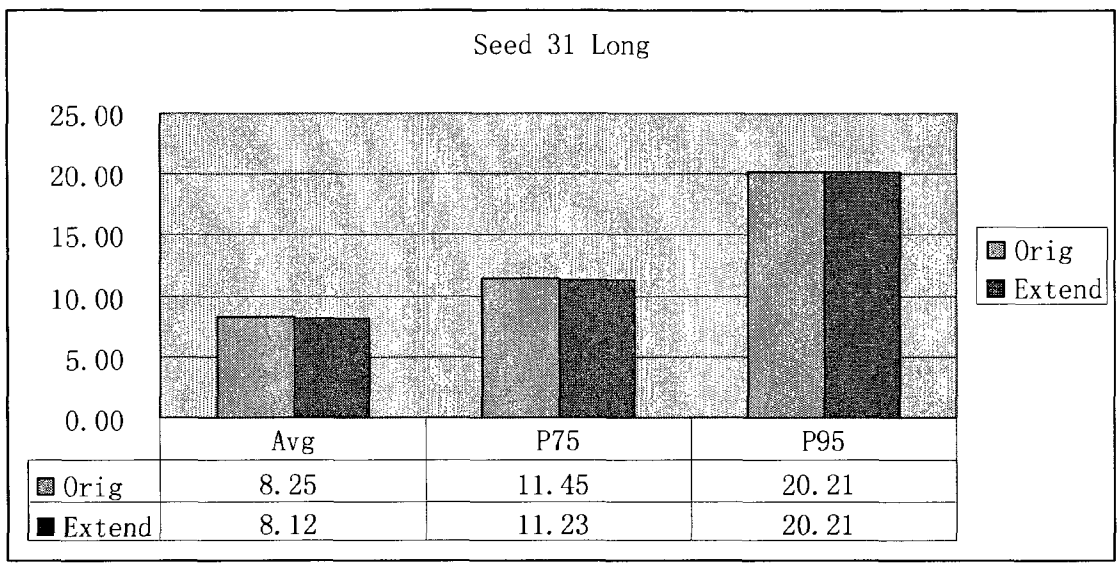
Figure 6.5 shows the comparison of relative response times for Seed 23 Long jobs between the Original result and our Extension.



	Avg	P75	P95
Orig	6.05	7.70	17.81
Extend	5.28	6.41	16.15

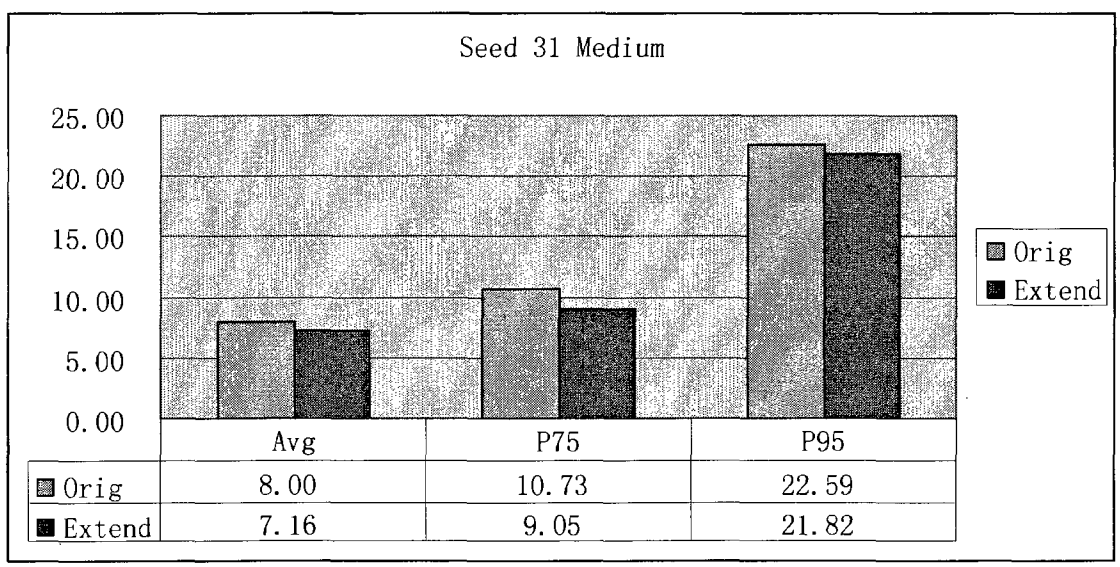
Figure 6.6: Seed 23 Medium jobs relative response time comparison

Figure 6.6 shows the comparison of relative response times for Seed 23 Medium jobs between the Original result and our Extension.



	Avg	P75	P95
Orig	8.25	11.45	20.21
Extend	8.12	11.23	20.21

Figure 6.7: Seed 31 Long jobs relative response time comparison

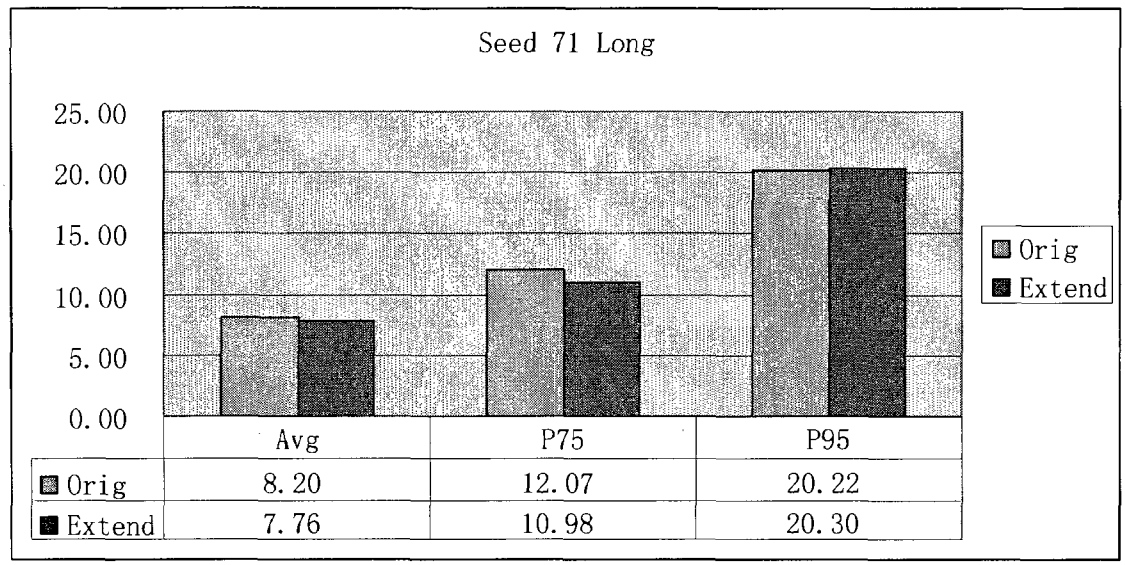


	Avg	P75	P95
Orig	8.00	10.73	22.59
Extend	7.16	9.05	21.82

Figure 6.8: Seed 31 Medium jobs relative response time comparison

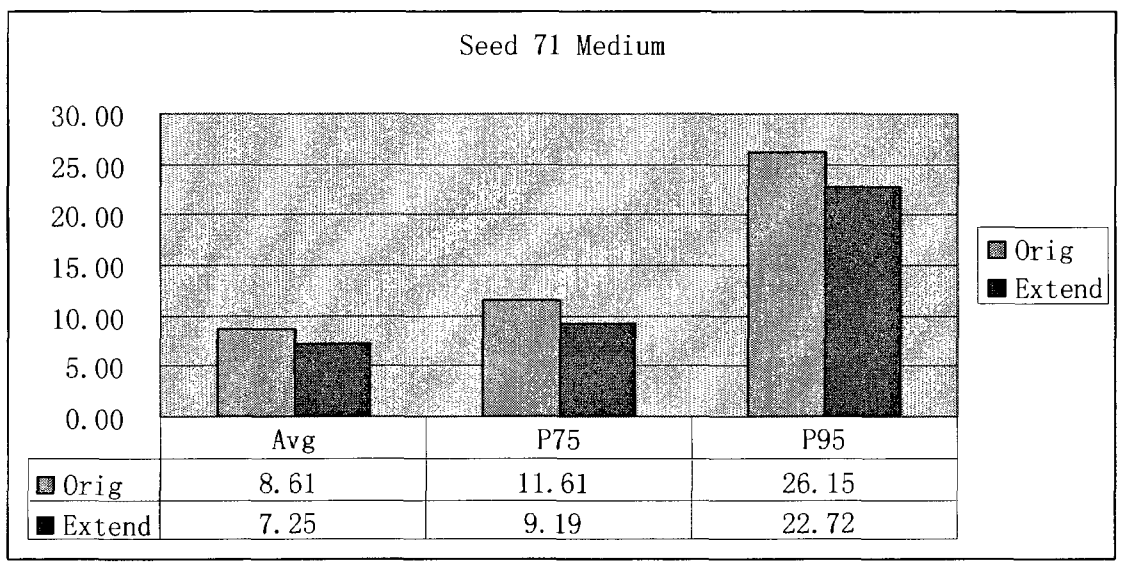
Figure 6.8 shows the comparison of relative response times for Seed 31 Long and Medium

jobs between the Original result and our Extension.



	Avg	P75	P95
Orig	8.20	12.07	20.22
Extend	7.76	10.98	20.30

Figure 6.9: Seed 71 Long jobs relative response time comparison



	Avg	P75	P95
Orig	8.61	11.61	26.15
Extend	7.25	9.19	22.72

Figure 6.10: Seed 71 Medium jobs relative response time comparison

Figure 6.10 shows the comparison of relative response times for Seed 31 Long and Medium jobs between the Original result and our Extension.

In summary, for all the five seeds we tested, the average response time for Long jobs are improved by 2.9%, the average response time for Medium jobs are improved by 13.1%. The result of P75 for Long jobs is improved by 1.9%, Medium jobs 20.0%. For P95 Long jobs, the improvement is 2.0% and for Medium jobs 9.0%. Our approach achieves improvements on medium job relative response times, and long job relative response times do not suffer any reduction.

As an explanation why long jobs do not benefit much: Scojo-PECT is a coarse-grain time sharing scheduler, jobs are classified by running time. Medium jobs are those runs from 10 minutes to 3 hours and Long jobs are now classified from 3 hours to up to over 50 hours. Approximately M slice interval is 18 minutes and L slice runs 42 minutes. When a long job migrates into M slice, it runs 18 extra minutes minus costs. When a medium job checkpoint and migrated into L slice it runs 42 extra minutes minus costs. So the extra run time for long migrated to medium could be relatively small. And it is very likely that the migrated M job will finish within the L slice. Moreover, the average job size of M jobs is smaller than L jobs this makes M jobs more flexible to be scheduled.

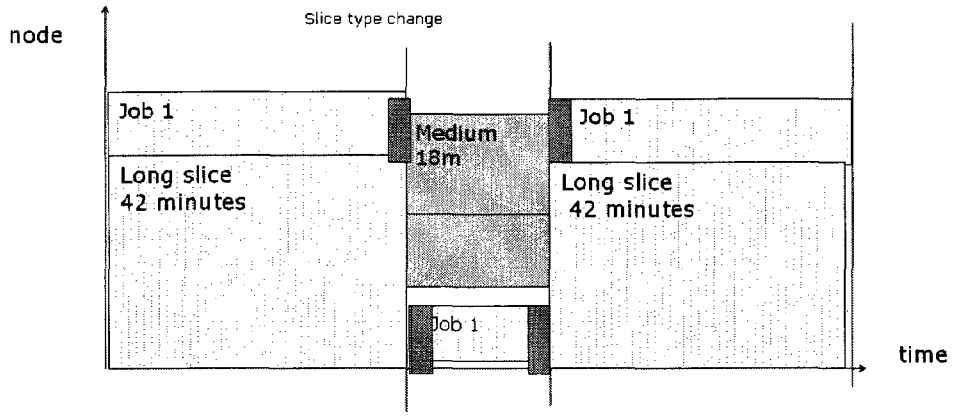


Figure 6.11: Long job migrate into M and migrate back to L

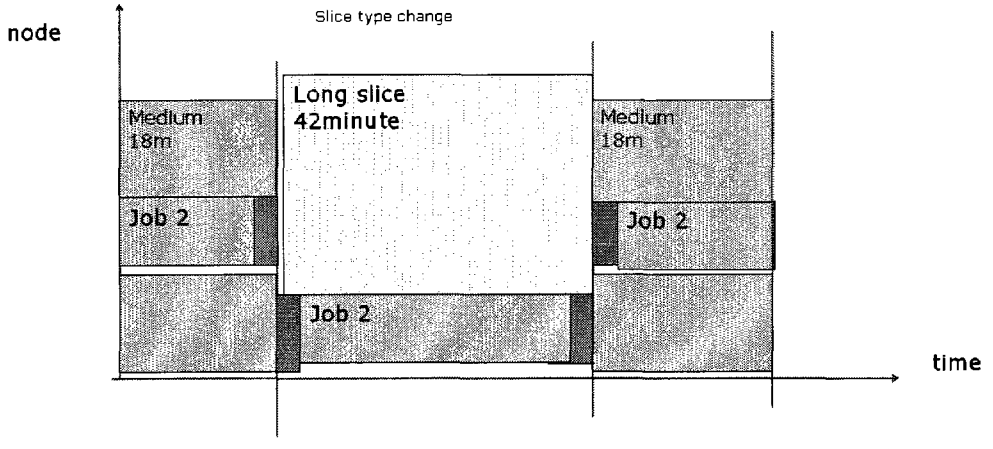


Figure 6.12: M job migrate into L and migrate back to M

7. Summary and Conclusion

We have presented our approach of extending the Scojo-PECT scheduler by Migration based on system-level checkpointing. The focus of our presented approach is to obtain improvement in average response times comparing to the original Scojo-PECT scheduler which is a coarse-grain time sharing framework.

We explored possible situations where checkpoint and migration can be applied and categorized them into four basic cases.

- We checkpoint job or multiple jobs and migrate them so that they can continue to run in next non-type slice.
- We checkpoint and migrate job or multiple jobs to make space in a time slice to make space for another job to perform non-type backfill.
- We let new arrived jobs to perform non-type backfill by checkpointing this job at the end of the slice and migrate it to avoid conflicts with next slice.
- We halt and remove non-type backfilled job(s) if new job of own type arrives.
- Checkpoint and migration cost and gain are calculated and evaluated to support decision making on which job or combination of jobs should be checkpoint and migrated.
- Heuristics are applied when making the decision which job or combination of jobs should be checkpoint and migrated.

Our extension as expected is able to reduce average relative response time of jobs. The experimental result shows that our approach improves average relative response time of Medium jobs by about 13.1% and improves about 2.9 % for Long jobs compared to the original Scojo-PECT scheduler without checkpointing and migration supported.

References

[1] S. Agarwal, G.S. Choi, C.R. Das, A.B. Yoo, S. Nagar, Co-ordinated coscheduling in time-sharing clusters through a generic framework, Proc. of the IEEE Intl. Conf. on Cluster Comp., 2003, pp. 84-91.

[2] A.C. Arpaci-Dusseau, D. Culler, Implicit Co-Scheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. ACM Trans. Compu. Sys., Aug. 2001,pp. 283-331.

[3] N. Bansal, M.H. Balter, Analysis of SRPT Scheduling: Investigating Unfairness, ACM SIGMETRICS Performance Evaluation Review, 2001, pp. 279-290.

[4] R. Brightwell, K.D. Underwood, C. Vaughan, An Evaluation of the Impacts of Network Bandwidth and Dual-Core Processors on Scalability, Proc. Internat. Supercomputing Conference, Dresden, Germany, Jun. 2007.

[5] L. Chai, Q. Gao, D. K. Panda, Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System, CCGRID, IEEE Inter. Symp. 2007, pp. 471-478

[6] S.P. Dandamudi, T.K. Thyagaraj, A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems, High Performance Comp. Proc. 4th Intl. Conf., 1997, pp. 218-223.

[7] B. Esbaugh, A.C. Sodan, Coarse-Grain Time Slicing with Resource-Share Control in Parallel-Job Scheduling, High Performance Computing and Communication (HPCC), Houston, LNCS 4782, Springer Verlag, Sept. 2007.

[8] D.G. Feitelson, A Survey of Scheduling in Multiprogrammed Parallel Systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[9] D.G. Feitelson, L. Rudolph, Gang Scheduling Performance Benefits for Fine-Grain Synchronization. Journal of Parallel and Distributed Computing, Dec 1992, pp. 306–318.

[10] E. Frachtenberg, D. G. Feitelson, F. Petrini, J. Fernandez, Adaptive Parallel Job Scheduling with Flexible Coscheduling, Parallel & Distributed Systems, IEEE Trans., 2005, pp. 1066–1077.

[11] F. Gine, F. Solsona, P. Hernandea, E. Luque, Adjusting Time Slices to Apply Coscheduling Techniques in a Non-dedicated NOW, Euro-Par 2002 Parallel Proc., 2002, pp. 234-239.

[12] F. Gine, F. Solsona, P. Hernandez, E. Luque, Cooperating Coscheduling in a Non-dedicated Cluster, Euro-Par 2003 Parallel Proc., 2003, pp. 212-217.

[13] F. Gine, F. Solsona, M. Hanzich, P. Hernandez, E. Luque, Cooperating Coscheduling: A Coscheduling Proposal Aimed at Non-Dedicated Heterogeneous NOWs, Journal of Computer Science and Technology, Vol. 22, Sep. 2007.

[14] D. G. Feitelson and M. A. Jette, "Improved utilization and responsiveness with gang scheduling". In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (Eds.), pp. 238-261, Springer-Verlag, 1997. Lecture Notes in Computer Science Vol. 1291

[15] M. Hanzich, F. Gine, P. Hernandez, F. Solsona, E. Luque, A Space and Time Sharing Scheduling Approach for PVM Non-dedicated Clusters, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2005, pp. 379-387.

[16] M. Hanzich, F. Gine, P. Hernandez, F. Solsona, E. Luque, CISNE: A New Integral Approach for Scheduling Parallel Applications on Non-dedicated Clusters, Euro-Par 2005 Parallel Processing, 2005, pp. 220–230.

[17] A. Hori, T. Yokota, Y. Ishikawa, S. Sakai, H. Konaka, M. Maeda, T. Tomokiyo, J. Nolte, H. Matsuoka, K. Okamoto, H. Hirono, Time Space Sharing Scheduling and architectural support, Job

Scheduling Strategies for Parallel Processing(JSSPP), 1995, pp. 92-105.

[18] U. Lublin, D.G. Feitelson, The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs, Journal of Parallel and Distributed Computing, 2003, pp. 1105-1122.

[19] A. Moursy, R. Garg, D.H. Albonesi, S. Dwarkadas, Compatible Phase Co-Scheduling on A CMP of Multi-Threaded Processors, Parallel & Distributed Proc. Sys., 20th IEEE Intl., 2006.

[20] T. Moscibroda, O. Mutlu, Memory Performance Attacks: Denial of Memory Service in Multi-core Systems. Proc. Of 16 th USENIX Security Symp., Boston, 2007.

[21] J.K. Ousterhout, Scheduling Techniques for Concurrent Systems, In 3rd Intl. Conf. Distributed Comput. Syst. (ICDCS), Oct 1982, pp. 22–30.

[22] F. Petrini, W. Feng, Buffered coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems, Parallel & Distributed Proc. Sys., 14th IEEE Intl., May 2000, pp. 439-444.

[23] G. Sabin, G. Kochhar, P. Sadayappan, Job Fairness in Non-Preemptive Job Scheduling, Proc. Intern. Conf. on Parallel Proc. IEEE, 2004.

[24] G. Sabin, V. Sahasrabudhe, On Fairness in Distributed Job Scheduling Across Multiple Sites, CLUSTER IEEE, 2004.

[25] G. Sabin, P. Sadayappan, Unfairness Metrics for Space-Sharing Parallel Job Schedulers, Job Scheduling Strategies for Parallel Processing, Springer, Vol. 3834, 2005.

[26] A. Snavely, D.M. Tullsen, G. Voelker, Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor, Proc. of the 2002 ACM SIGMETRICS Intl. Conf. on Measurement & Modeling of Computer Systems, Jun. 2002.

[27] A.C. Sodan, L. Lan, LOMARC — Lookahead Matchmaking for Multi-resource Coscheduling, Job Scheduling Strategies for Parallel Processing (JSSPP), 2005, pp. 288-315.

[28] A.C. Sodan, G. Gupta, A. Deshmeh, X. Zeng, Benefits of Semi Time Sharing and Trading Space vs. Time in Computational Grids, Technical Report 08-020, University of Windsor, Computer Science, May 2008.

[29] A.C. Sodan, L. Lan, LOMARC Lookahead Matchmaking for Multiresource Coscheduling on Hyperthreaded processors, Parallel and Distributed Systems, IEEE Trans., 2006, pp.1360-1375.

[30] A.C. Sodan, Adaptive Scheduling for QoS Virtual-Machines under Different Resource Availability—First Experiences, Job Scheduling Strategies for Parallel Processing (JSSPP), 2009.

[31] A.C. Sodan, Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey, Concurrency & Computation: Practice & Experience, 17(15), Dec. 2005, pp. 1725-1781.

[32] Sriram Krishnan, Dennis Gannon: Checkpoint and Restart for Distributed Components in XCAT3. GRID 2004: 281-288. 10, Electronic Edition

[33] D.M. Tullsen, A. Snavely, Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, Internat. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[34] D. Lifka. The ANL/IBM SP scheduling system. In JSSPP, pages 295–303, 1995.

[35] J. Weinberg, A. Snavely, Symbiotic Space-Sharing on SDSC's DataStar System, Job Scheduling Strategies for Parallel Processing (JSSPP), 2007, Vol. 4376. pages 295–303, 1995.

[36] A.B. Yoo, M. A. Jette, An Efficient and Scalable Coscheduling Technique for Large Symmetric Multiprocessor Clusters, Job Scheduling Strategies for Parallel Processing (JSSPP), 2001, pp. 21-40.

[37] J.L. Yu, D. Azougagh, J.S. Kim, S.R. Maeng, PROC Process ReOrdering-Based Coscheduling on Workstation Clusters, Parallel & Distributed Proc. Sys.,19th IEEE Intl., Apr. 2005, pp. 50-50.

[38] X. Zeng, J. Shi, X. Cao, A.C. Sodan, Grid Scheduling with ATOP-Grid under Time Sharing, CoreGrid Workshop on Grid Middleware (in conjunction with ISC), Dresden, Springer, Jun. 2007.

[39] Y. Zhang, A. Sivasubramaniam, Scheduling Best-effort and Real-time Pipelined Applications , on Time-Shared Clusters, Proc. of the 13th annual ACM Sys. on parallel algorithms and architectures (SPAA), July 2001.

[40] B.B Zhou, R.P. Brent, On the Development of an Efficient Coscheduling System, Job Scheduling Strategies for Parallel Procesing (JSSPP), Springer, 2001, pp. 103-115.

[41] Jones JP, Nitzberg B. Scheduling for parallel Supercomputing: A historical perspective on achievable utilization.

[42] Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (Lecture Notes in Computer Science, vol. 1659). Springer: Berlin, 1999.

[43] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi , Yi-Min Wang, David B. Johnson A survey of rollback-recovery protocols in message-passing systems.ACM Computing Surveys (CSUR) Volume 34, Issue 3, Pages: 375 – 408, 2002

[44] Luís Moura Silva, João Gabriel Silva: An Experimental Evaluation of Coordinated Checkpointing in a Parallel Machine. EDCC 1999: 124-142

[45] Sankaran S, Squyres J M, Barrett B, Lumsdaine A, Duell J, Hargrove P and Roman E 2003 The LAM/MPI checkpoint/restart framework: system-initiated checkpointing Proc. Los Alamos Computer Science Institute (LACSI) Symp. (Santa Fe, New Mexico, USA, October 2003)

[46] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kal´ e. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In 2004 IEEE International Conference onCluster Computing, pages 93–103, San Dieago, CA, September 2004Gengbin

[47] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. Proceedings of the 10th International Parallel Processing Symposium, p. 526-531,

[48] Hua Zhong and Jason Nieh http://www.ncl.cs.columbia.edu/research/migrate/crak.html

[49] Gioiosa, R. Sancho, J.C. Jiang, S. Petrini, F. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference Publication Date: 12-18 Nov. 2005

[50] U.Lublin, D.G. Feitelson, "The Workload on parallel Super computers: Modeling the Characteristics of Rigid Jobs", Journal of Parallel and Dis tributed Computing, Vol. 63, No.11, Nov 2003, pp.1105-1122

[51] José Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernández, Eitan Frachtenberg: On the Feasibility of Incremental Checkpointing for Scientific Computing. IPDPS 2004

[52] "Parix 1.2: Software Documentation", Parsytec Computer GmbH, March 1999

Appendix

In this appendix, we put our result of test cases during our implementation. We created jobs manually and simulated the situations that individual cases should be handled.

Case 1:

For case 1

We first tested M job checkpoint and migrate to L slice. Our input is

Job1 using 128 nodes and run 299 seconds.

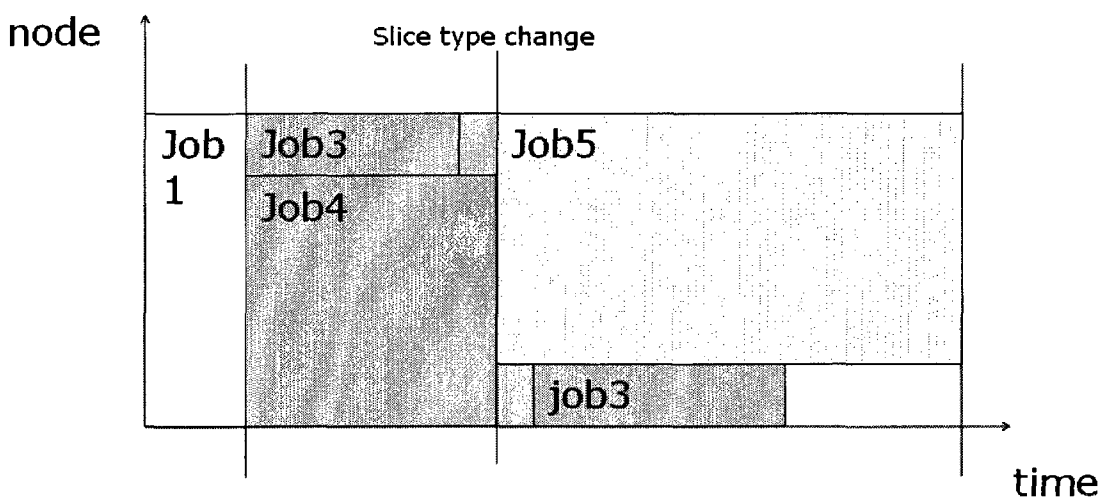
Job2 using 128 nodes and run 299 seconds.

Job3 using 28 nodes and run 8800 seconds

Job4 using 100 nodes and run 9000 seconds

Job5 using 100 nodes and run 20000 seconds.

Job1 and 2 are short jobs, Job 3 and 4 are medium jobs, Job5 is a long job. Our test result shows that job3 is migrated to Long slice and run together with Job5 as expected.



Then we tested L job checkpoint and migrate to M slice. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 1 node and run 1000 seconds.

Job4 using 128 nodes and run 299 seconds

Job5 using 127 nodes and run 9000 seconds

Job6 using 100 nodes and run 20000 seconds

Job7 using 1 node and run 20000 seconds.

Job 1, 2 and 4 are short jobs, Job 3 and 5 are medium jobs, Job6 and 7 are long jobs. Our test result shows that job7 is migrated to Medium slice and run together with Job5 as expected.

We tested multiple jobs checkpoint and migrate. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 14 nodes and run 8800 seconds.

Job4 using 100 nodes and run 9000 seconds

Job5 using 14 nodes and run 8800 seconds

Job6 using 100 nodes and run 20000 seconds

Job1 and 2 are short jobs, Job 3, 4 and 5 are medium jobs, Job6 is a long jobs. Our test result shows that job 3 and 5 are migrated to Long slice and run together with Job6 as expected.

Case 2

For case 2

We first tested single medium job migrate and allow backfill. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

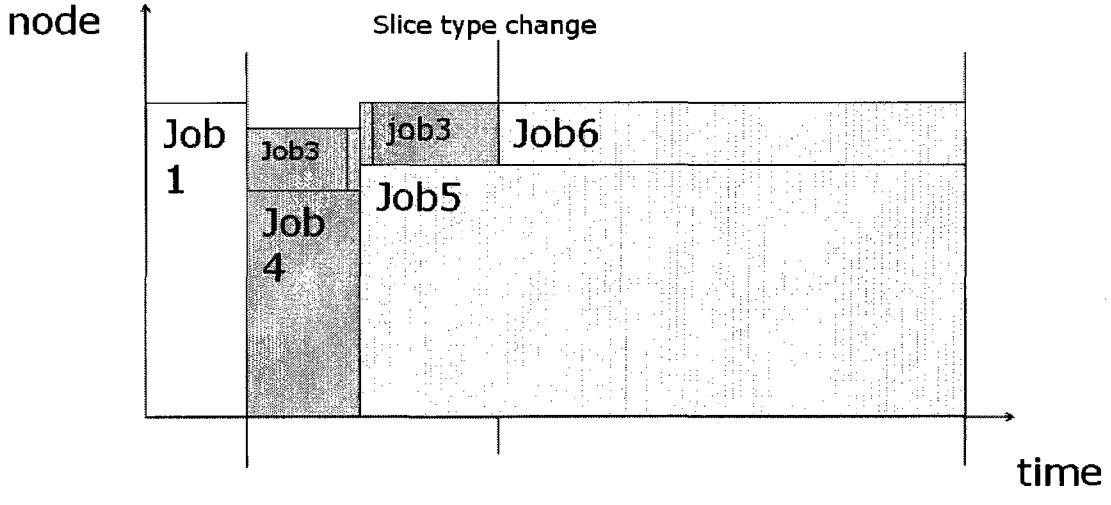
Job3 using 29 nodes and run 1700 seconds

Job4 using 99 nodes and run 1300 seconds

Job5 using 28 nodes and run 20000 seconds.

Job6 using 100 nodes and run 20000 seconds.

Job1 and 2 are short jobs, Job 3 and 4 are medium jobs, Job5 and 6 are long jobs. Our test result shows that after job 4 finished, job3 is migrated to free nodes in current slice and job 5 is non-type backfilled and run.



Next we tested multiple jobs migrate and allow backfill. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 14 nodes and run 1700 seconds

Job4 using 15 nodes and run 1700 seconds

Job5 using 99 nodes and run 1300 seconds

Job6 using 28 nodes and run 20000 seconds.

Job7 using 100 nodes and run 20000 seconds.

Job1 and 2 are short jobs, Job 3, 4 and 5 are medium jobs, Job6 and 7 are long jobs. Our test result shows that after job 5 finished, job3 and 4 are migrated to free nodes in current slice and job 6 is non-type backfilled and run.

We then tested single long job migrate and allow backfill. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 28 nodes and run 2500 seconds

Job4 using 100 nodes and run 1700 seconds

Job5 using 28 nodes and run 4600 seconds.

Job6 using 100 nodes and run 20000 seconds.

Job1 and 2 are short jobs, Job 3, 4 and 5 are medium jobs, Job 6 is a long jobs. Our test result shows that after job 4 finished in long slice, job6 is migrated to free nodes in current slice and job 3 is non-type backfilled and run.

Case 3

For case 3, we first tested allow new long job to backfill in Long slice. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

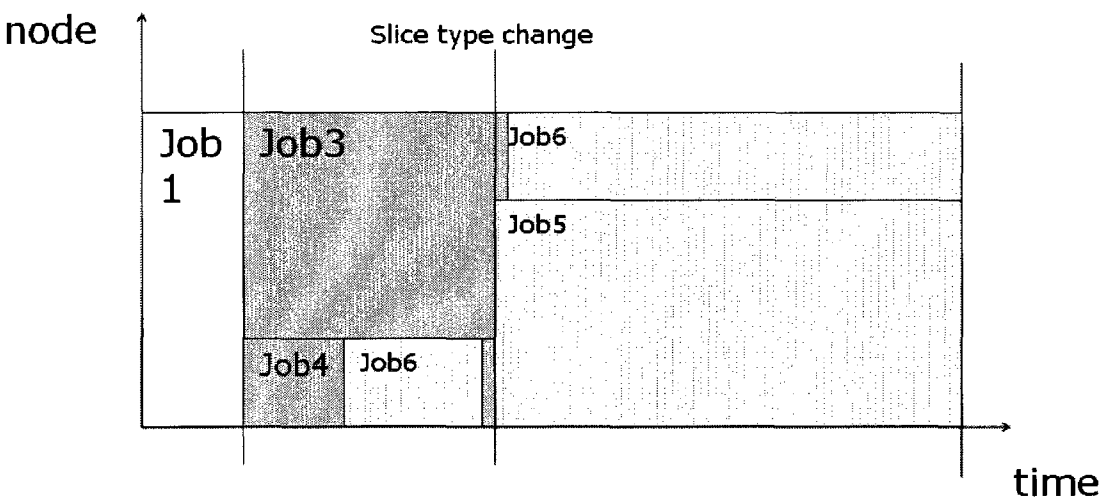
Job3 using 100 nodes and run 1700 seconds

Job4 using 28 nodes and run 1300 seconds

Job5 using 100 nodes and run 20000 seconds.

Job6 using 28 nodes and run 20000 seconds.

Job1 and 2 are short jobs, Job 3 and 4 are medium jobs, Job5 and 6 are long jobs. Our test result shows that after job 4 finished, Job6 allowed to non-type backfill and checkpoint-migrated in Long slice to avoid conflict with Job5



Then we tested allow multiple jobs to backfill and then checkpoint and migrate. Our input is:

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 100 nodes and run 1700 seconds

Job4 using 28 nodes and run 1300 seconds

Job5 using 100 nodes and run 20000 seconds.

Job6 using 14 nodes and run 20000 seconds.

Job7 using 14 nodes and run 20000 seconds.

Job1 and 2 are short jobs, Job 3 and 4 are medium jobs, Job5, 6 and 7 are long jobs. Our test result shows that after job 4 finished, Job6, 7 allowed to non-type backfill and checkpoint-migrated in Long slice to avoid conflict with Job5

Then we tested allow medium jobs to backfill and then checkpoint and migrate. Our input is:

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 100 nodes and run 1700 seconds

Job4 using 29 nodes and run 2000 seconds

Job5 using 99 nodes and run 20000 seconds.

Job6 using 28 nodes and run 2000 seconds.

Job1 and 2 are short jobs, Job 3, 4 and 6 are medium jobs, and Job5 is a long job. Our test result shows that after job 4 finished, Job6 is allowed to non-type backfill and checkpoint-migrated in M slice to avoid conflict with Job3

Case 4

For case 4, we first tested single job checkpointed out in M slice. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

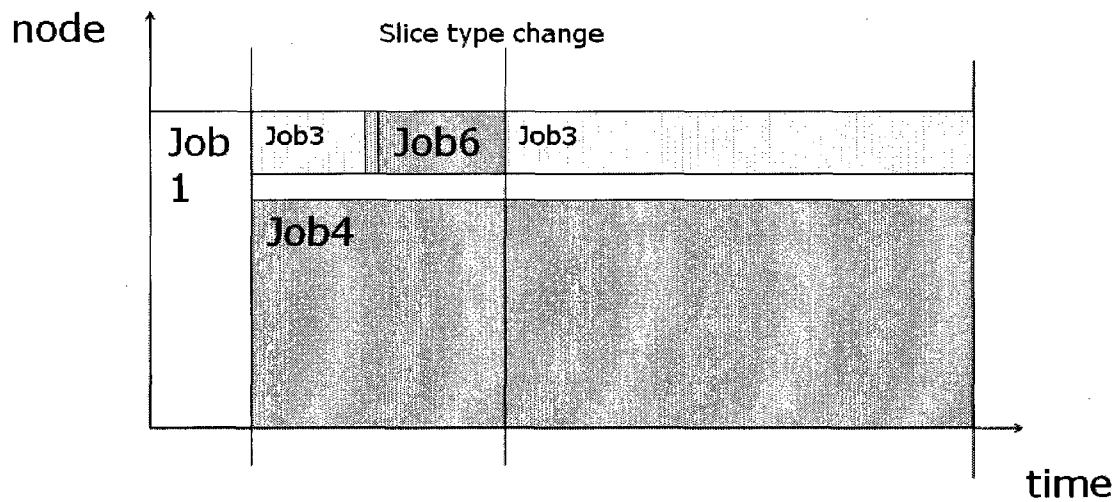
Job3 using 28 nodes and run 17000 seconds

Job4 using 99 nodes and run 1300 seconds

Job5 using 1 node and run 80 seconds.

Job6 using 100 nodes and run 2000 seconds.

Job1, 2 and 5 are short jobs, Job 4 and 6 are medium jobs, Job 3 is a long job. Our test result shows that after Job 6 submitted, the non-type backfilled job3 is checkpointed and preempted. Job 6 gets its nodes to run, when its Long slice, Job 3 gets back to run.



Then we tested multiple jobs checkpointed out to make space for own new arriving job. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 14 nodes and run 17000 seconds

Job4 using 14 nodes and run 17000 seconds

Job5 using 99 nodes and run 1300 seconds

Job6 using 1 node and run 80 seconds.

Job7 using 100 nodes and run 2000 seconds.

Job1, 2 and 6 are short jobs, Job 5 and 7 are medium jobs, Job 3, 4 are long jobs. Our test result shows that after Job 7 submitted, the non-type backfilled job3, 4 are checkpointed and preempted. Job 7 gets its nodes to run, when its Long slice, Job 3, 4 gets back to run.

Then we tested single job checkpointed out to make space for own new arriving job in Long slice. Our input is

Job1 using 128 nodes and run 299 seconds.

Job2 using 128 nodes and run 299 seconds.

Job3 using 128 nodes and run 299 seconds

Job4 using 28 nodes and run 17000 seconds

Job5 using 100 nodes and run 18000 seconds.

Job6 using 1 nodes and run 80 seconds

Job7 using 90 nodes and run 1800 seconds

Job1, 2, 3 and 6 are short jobs, Job 7 is a medium job, Job 4 and 5 are long jobs. Our test result shows that after Job 7 submitted, the non-type backfilled job5 is checkpointed Job 7 gets its nodes to run, when its Long slice, Job 5 gets back to run.

Vita Auctoris

NAME:	Peiyu Cai
PLACE OF BIRTH:	Hebei, P.R. China
YEAR OF BIRTH:	1980
EDUCATION:	Hebei University of Technology, Hebei, China 1999 – 2003 University of Windsor, Windsor, Ontario, Canada 2006 – 2009